PDC Summer School 2010
Lab wrap-up: Serial code efficiency

August 21, 2010

The first part (multiplying a matrix my a scalar) should be fairly self-explanatory. Here we focus on the matrix-matrix multiplication tasks.

1 Basic implementation and base-line efficiency analysis

A basic implementation of matrix-matrix multiplication could be as follows (in C only) in Listing 1, with results in 1 (left).

Listing 1: Basic MATMUL

```
void mul(double* c, const double* a, const double* b, int N)
{
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
    {
      double s = 0;
      for(int k=0; k<N;k++)
      s += a[i*N+k]*b[j+k*N];
      c[i*N+j] = s;
    }
}
```

We note several things here:

- The inner loop has stride `N` when loading from `b`. This puts us at risk for poor cache utilization.
- Adding the `restrict` keyword to all pointer declarations (e.g. `const double* restrict b`) creates an opportunity for the compiler to optimize further at level `-O3`. These results are seen in Figure 1 as well.
- The basic implementation gets around 1 Gflop/s for small matrices and 200 Mflop/s for bigger ones. With optimization the result is roughly 2.4 Gflop/s for small and 800 Mflop/s for big.
- The present hardware should be able to do at least two floating point operations per clock cycle, i.e. 5.34 Gflop/s. That is, the base implementation runs no faster than 1/25 of peak flops – clearly unacceptably slow.
- We use PAPI to analyze what is going on in the range $N = 300 \ldots 500$:

```
N = 300
PAPI_FP_OPS ................................     5.47977e+07
PAPI_L1_DCM ................................     3.62696e+06
PAPI_L2_TCM ................................           32985

N = 500
```

```
PAPI_FP_OPS ..................................      2.51269e+08
PAPI_L1_DCM ..................................      1.51698e+08
PAPI_L2_TCM ..................................            26371
```

We note that the number of L1 cache misses is a full order less than the number of floating point operations for the case $N = 300$, whereas they these numbers are almost equal for $N = 500$. The drop in performance is caused by a sharp increase in the number of L1 cache misses.

- For the `-O3` optimized code the drop occurs between $N = 700$ and 1000. Again with papiex:

```
N = 700
PAPI_FP_OPS ..................................      3.47242e+08
PAPI_L1_DCM ..................................      4.31669e+07
PAPI_L2_TCM ..................................           375733

N = 1000
PAPI_FP_OPS ..................................      1.00475e+09
PAPI_L1_DCM ..................................      1.25777e+08
PAPI_L2_TCM ..................................      9.83896e+07
```

Here, the relative increase in the number of L2 cache misses is causing the drop in performance.

Listing 2: MKL wrapper

```
void mul(double* c, const double* a, const double* b, int N)
{
  /* parameters for C <- A*B, N-by-N */
  const CBLAS_ORDER order = CblasRowMajor;
  const CBLAS_TRANSPOSE transA = CblasNoTrans;
  const CBLAS_TRANSPOSE transB = CblasNoTrans;
  const double alpha = 1.0;
  const double beta = 0.0;
  const int lda = N;
  const int ldb = N;
  const int ldc = N;

  cblas_dgemm(order, transA, transB, N, N, N, alpha,
              a, lda, b, ldb, beta, c, ldc);
}
```

2 Vendor-tuned library

In Listing 2 we give a typical way to call the BLAS function DGEMM from C, and performance results in Figure 1. This function is incredibly well tuned, and should be used whenever available! Note: Intel MKL is free for non-commercial purposed on Linux. The fastest free BLAS implementation is typically "GOTO BLAS"[1]

3 Beating the compiler – manual optimization

To improve the base implementation, we need to deal with the poor reuse of data. First, however, note that permuting the loop order $(i, j, k) \to (k, i, j)$ and unrolling the middle loop by four improves the situation considerably:

[1] `http://www.csar.cfs.ac.uk/user_information/software/maths/goto.shtml`

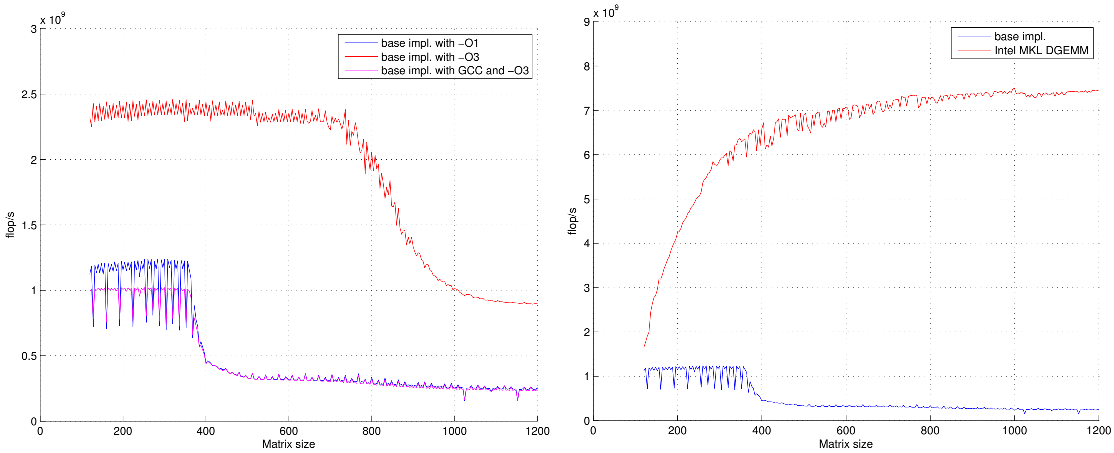

Figure 1: Left: performance of basic MATMUL, different compilers and optimization levels. For -O3 optimization, the "restrict" keyword was added in the code. Right: Basic MATMUL vs. Intel MKL CBLAS DGEMM

Listing 3: Unrolled and reordered implementation of MATMUL

```
void mul(double* restrict c,
         const double* restrict a,
         const double* restrict b, int N)
{
  double s0, s1, s2, s3, r;
  for(int k=0; k<N; k++)
    for(int i=0; i<N; i+=4)
    {
      s0 = a[     i*N+k];
      s1 = a[(i+1)*N+k];
      s2 = a[(i+2)*N+k];
      s3 = a[(i+3)*N+k];
      for(int j=0; j<N; j++)
      {
        r = b[k*N+j];
        c[     i*N+j] += s0*r;
        c[(i+1)*N+j] += s1*r;
        c[(i+2)*N+j] += s2*r;
        c[(i+3)*N+j] += s3*r;
      }
    }
}
```

Note that the inner loop has unit stride in both the one load and four store operations. It appears that the compiler managed to do essentially this with `-O3` optimization.

To improve data locality we block all three loops as suggested, see Listing 4 and Figure 2 (left). An additional level of blocking may be beneficial, so that both L1 and L2 cache can be well utilized. This implementation is given in Listing 5 and results in Figure 2 (right). Here we have combined the blocking with the loop permutation and unrolling. Some observations:

- The blocked versions do not suffer the decrease in efficiency for larger matrices that was seen with the basic implementation. They beat the compiler by a significant factor.
- The second blocked version runs at about 50% of peak performance. However, the compiler did not generate SSE instructions for the inner loop (which could in theory make the code run twice as fast).

- What about the dips at a few matrix sizes? These seem to occur at conspicuous values: roughly $N = 256, 384, 512, 640, 768, 896, 1024, 1152$, all multiples of 128. The dips are caused by conflict misses, i.e. that most load operations are mapped into the same cache bank.

Listing 4: First blocked implementation of MATMUL

```
void mul(double* restrict c, const double* a, const double* b, int N)
{
  double s0,s1,s2,s3,r;
  int i,j,k;
  int i0,j0,k0;
  /* block loop */
  for(i0=0; i0<N; i0+=BLOCK_SIZE)
    for(j0=0; j0<N; j0+=BLOCK_SIZE)
      for(k0=0; k0<N; k0+=BLOCK_SIZE)
        /* mul loop */
          for(k=k0; k<MIN(k0+BLOCK_SIZE,N); k++)
            for(i=i0; i<MIN(i0+BLOCK_SIZE,N); i+=4)
            {
              s0 = a[ i   *N+k];
              s1 = a[(i+1)*N+k];
              s2 = a[(i+2)*N+k];
              s3 = a[(i+3)*N+k];
              for(j=j0; j<MIN(j0+BLOCK_SIZE,N); j++)
                {
                  r = b[k*N+j];
                  c[ i   *N+j] += s0*r;
                  c[(i+1)*N+j] += s1*r;
                  c[(i+2)*N+j] += s2*r;
                  c[(i+3)*N+j] += s3*r;
                }
              }
}
```

Listing 5: Twice blocked, unrolled MATMUL

```
void mul(double* restrict dest,
         const double* restrict a,
         const double* restrict b, int N)
{
  double s0,s1,s2,s3;
  int i,j,k;
  int i0,j0,k0;
  int i1,j1,k1;
  /* block loop L2*/
  for(i0=0; i0<N; i0+=BLOCK_SIZE)
    for(j0=0; j0<N; j0+=BLOCK_SIZE)
      for(k0=0; k0<N; k0+=BLOCK_SIZE)
        /* block loop L1 */
        for(i1=i0; i1<i0+BLOCK_SIZE; i1+=BLOCK_SIZE_II)
          for(j1=j0; j1<j0+BLOCK_SIZE; j1+=BLOCK_SIZE_II)
            for(k1=k0; k1<k0+BLOCK_SIZE; k1+=BLOCK_SIZE_II)
              /* mul loop */
              for(i=i1; i<MIN(i1+BLOCK_SIZE_II,N); i++)
                for(j=j1; j<MIN(j1+BLOCK_SIZE_II,N); j+=4)
                {
                  s0 = 0, s1 = 0, s2 = 0, s3 = 0;
                  for(k=k1; k<MIN(k1+BLOCK_SIZE_II,N); k++)
                  {
```

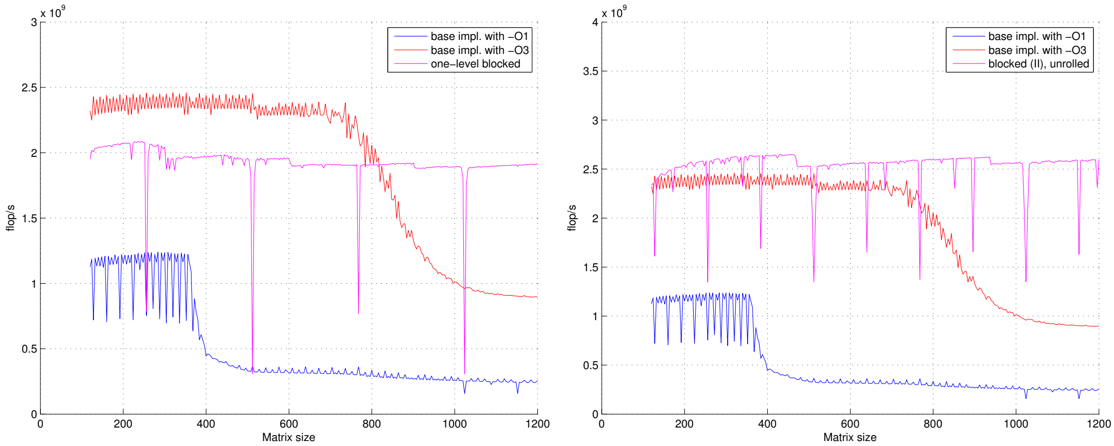

Figure 2: Left: performance of blocked MATMUL vs. base implementation, Right: performance of two-level blocked MATMUL vs. base implementation

```
                    s0 += a[i*N+k]*b[j+k*N];
                    s1 += a[i*N+k]*b[j+1+k*N];
                    s2 += a[i*N+k]*b[j+2+k*N];
                    s3 += a[i*N+k]*b[j+3+k*N];
                  }
                  dest[i*N+j  ] += s0;
                  dest[i*N+j+1] += s1;
                  dest[i*N+j+2] += s2;
                  dest[i*N+j+3] += s3;
                }
}
```