Optimizing Single Core Matrix Multiply

Group 2
September 17, 2015

1 Intro

Matrix multiplication is a ubiquitous operation in science and engineering, and thus the efficient and fast computation of matrix products is essential. This report serves as an initial report into tuning the performance of matrix multiplication on a single core.

1.1 Matrix Multiplication

Throughout this report, we will consider the problem of computing $C = AB$, where C, A, B are $M \times M$ square matrices.

Listing 1: Naive Square Matrix Multiply

```
void square dgemm(const int M,
                  const double *A, const double *B, double *C)
{
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < M; ++j) {
            for (int k = 0; k < M; ++k) {
                C[j*M+i] += A[k*M+i] * B[j*M+k];
            }
        }
    }
}
```

1.2 Testing specs

2 Optimization strategies

At this time, two different optimization strategies have been implemented and tested:

2.1 Loop ordering

Notice that in the naive matrix multiply implementation, the innermost loop is where all the work is done. Thus, the order of the loops does not matter. Noting this, we can choose the loop order that best regularizes memory access. Ideally, we would like to access the arrays with stride 1, and this motivates the idea that the 'i' variable in the above naive code should be looped over in the innermost loop, as then we are accessing C and A with stride one. This results in the following code:

Listing 2: Improved Loop Order Square Matrix Multiply

```
void square_dgemm(const int M,
                  const double *A, const double *B, double *C)
{
    for (int j = 0; j < M; ++j) {
        for (int k = 0; k < M; ++k) {
            double bkj = B[j*M+k];
            for (int i = 0; i < M; ++i)
                C[j*M+i] += A[k*M+i] * bkj;
        }
    }
}
```

This optimization is supported empirically, as all possible loop orders were tested (see Figure 2.1).

This basic optimization brings the naive matrix multiple up to the speed of the provided Fortran code (see Figure **??**).

2.2 Block and Copy Optimization

In order to encourage cache reuse, and thus reducing the overhead of loading data from main memory when performing the floating point operations, it may make sense to partition the matrices

3 Future Work

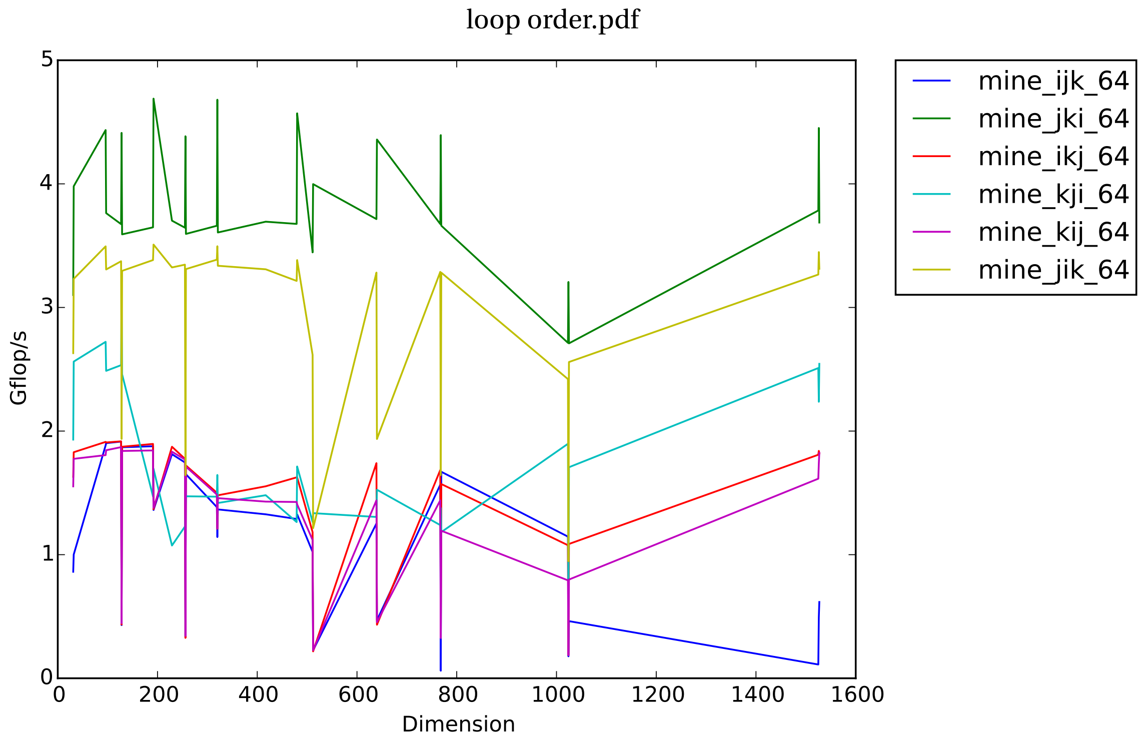

Figure 2.1: Results for all possible loop orders

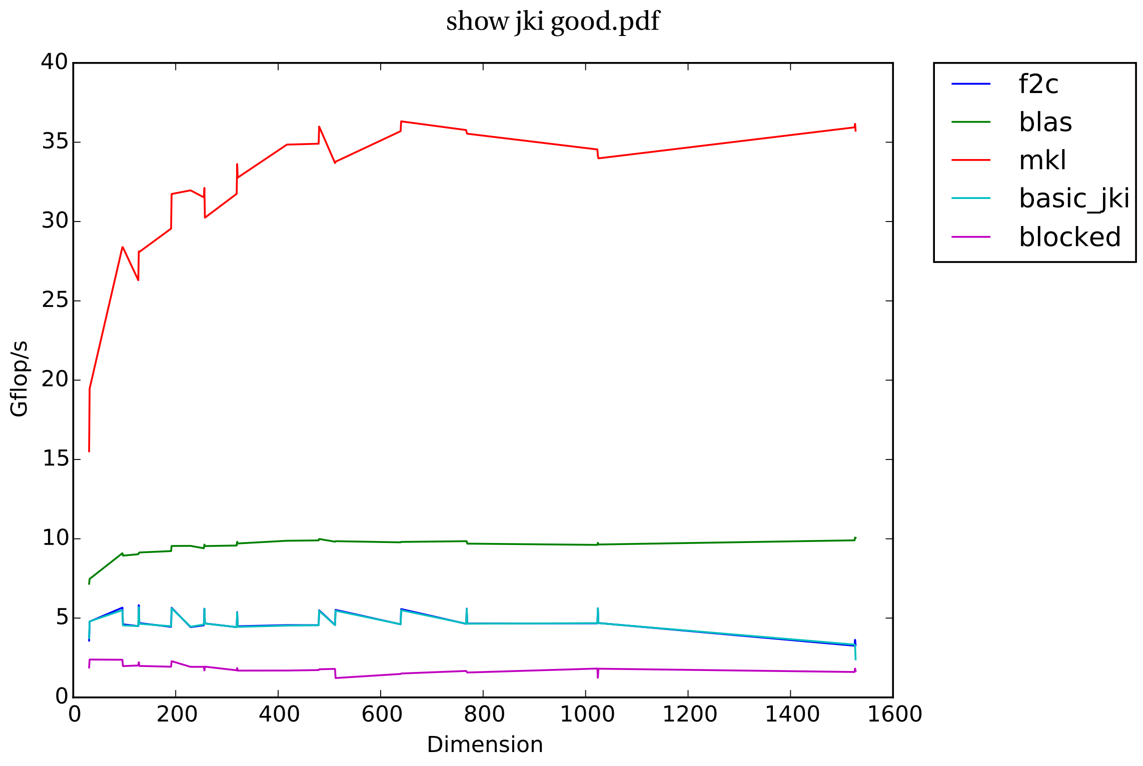

Figure 2.2: Loop order optimization comparison