Project 1 Write-Up, CS 5220

Hang Chu, Mohammed sameed Shafi, Lingnan Liu Jilong Wu, Sheng Wang

February 14, 2014

1 Optimizations Used

We used the following optimizations tricks:

- SSE coding
- blocking
- copy optimization
- compiler options

1.1 SSE coding

We started with the SSE coding functions provided on BitBucket, we adopted the (2-by-p)*(p-by-2) matrix multiplication subroutine. Testing with ktimer showed significant performance improvement: the original kdgemm function (with all size set as 4) gave approximately 2000mflops/s, while the SSE kernel gave 3500-4000mflop/s, which was what we expected as the SSE kernel should be two times faster because it computes two double number operations in parallel.

1.2 blocking

We break matrix A in to blocks of 2-by-p, matrix B into blocks of p-by-2, and matrix C into blocks of 2-by-2.

1.3 copy optimization

We noted that in order to use the SSE kernel, the 2-by-p matrix should be in column major and the p-by-2 matrix should be in row major. Thus, we add two $O(n^2)$ copy optimization steps to rearrange A into Ak and B into Bk. In Ak, the data storage order is [A_{11}, A_{21}, A_{12}, A_{22},...,A_{1n}, A_{2n}, A_{31}, A_{41},...], and in Bk the order is [B_{11}, B_{12}, B_{22}, B_{22},...,B_{n1}, B_{n2}, B_{13}, B_{14},...].
In that way, every time we call the SSE kernel, the 2-by-p and p-by-2 matrices are both in continuous storage, which guarantees high cache efficiency.
For matrices with size of odd number, we add a row and a column of all 0s two transfer them into matrices with size of even number. The operation is still $O(n^2)$ so it won't affect the performance too much.
Finally we rearrange the matrix Ck into C such that all elements are placed correctly.

1.4 compiler options

As the final step we tried different compiler options. We discovered that for our strategy, changing OPTFLAG -O3 to -O2 helps a lot. We are able to achieve 3000mflops/s by using -O2, which is approximately 75%-200% faster than dgemm_basic. We also tried adding flags such as -prefetch, -unroll-aggressive, and -SSE4.2, we didnot observe significant performance improvement with the first two flags, with -SSE4.2 we are able to make the result better with approximately 100mflops/s improvement.
In Figure 1-3 we show the comparisons.

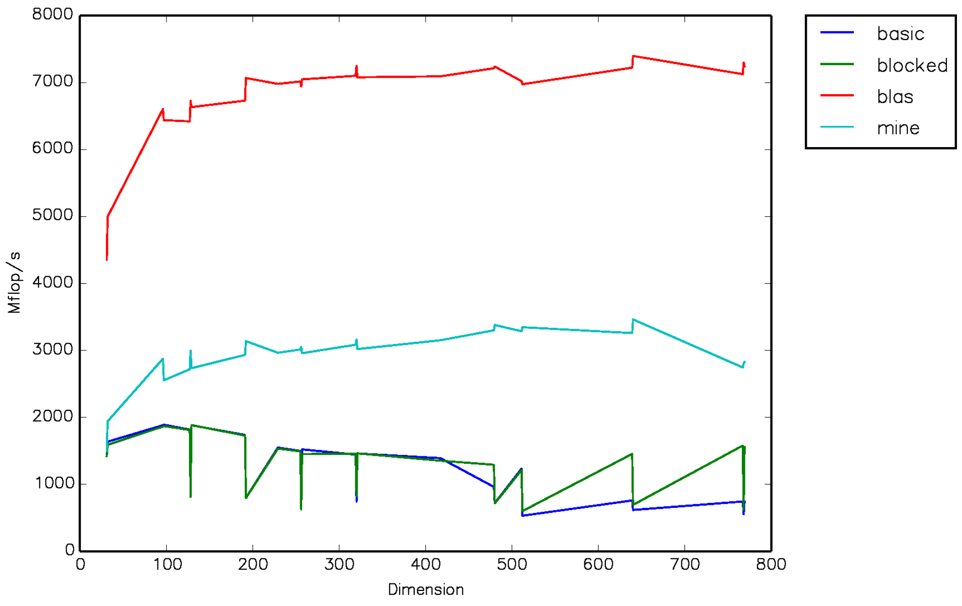

Figure 1: Results using -O2 -SSE4.2 flag.(**Our best result**)

Table 1: Results of our function.

size	31	32	96	97	127	128	129	191	192
mflops/s	1500.9	1941.36	2876.53	2553.52	2718.48	2997.99	2734.73	2933.85	3138.92
size	229	255	256	257	319	320	321	417	479
mflops/s	2965.58	3013.41	3053.13	2958.54	3085.47	3164.43	3020.34	3152.01	3300.3
size	480	511	512	639	640	767	768	769	
mflops/s	3380.07	3286.66	3346.75	3259.09	3462.89	2745.71	2807.68	2826.75	

2 Results

Our best result is reported in Figure 1, with the following specific mflop/s values in Table 1.

3 Explanations & Discussions

Generally, our method outperforms the basic one by more than 150%, there are two contributing factors: first our SSE kernel operates on two double number operations simultaneously, which improves the performance by approximately one time; second our copy optimization guarantees that every time the kernel is called, all the variables passed in are continuous in the memory, which significantly improves cache efficiency.

There are several exeptions: 1) When the size of matrix is relatively small (less than 100), the kernel operates on less numbers than the cache can hold, thus maximum cache efficiency is not reached. 2) When the size of matrix becomes too large, the cache can not hold all the data each time we call the kernel, so the performance decreases. 3) When the size of matrix is an odd number, an extra rearranging operation to fill an all 0 row and column is performed, that explains why things like size 127 being slower than size 128 happen.

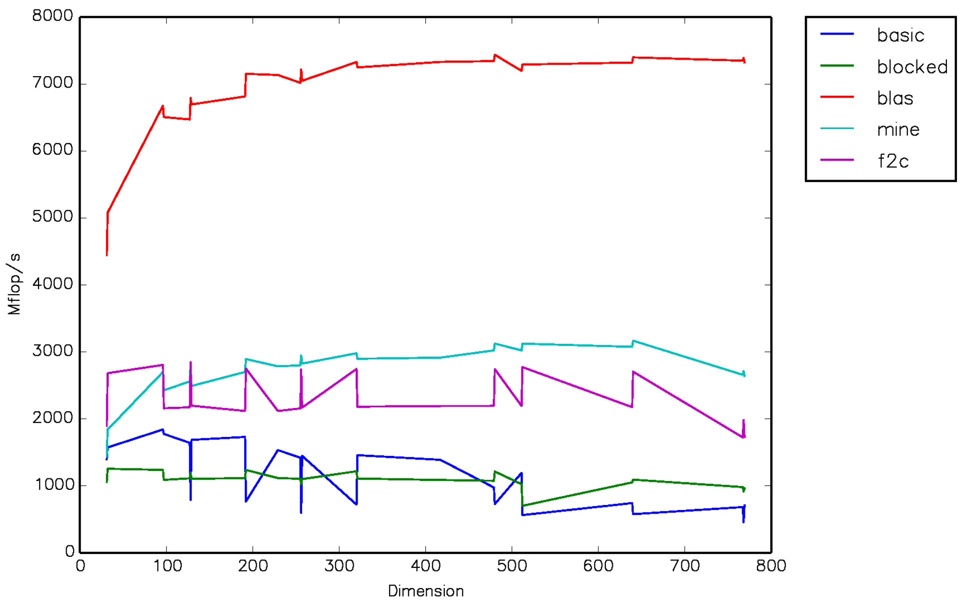

Figure 2: Results using only -O2 flag.

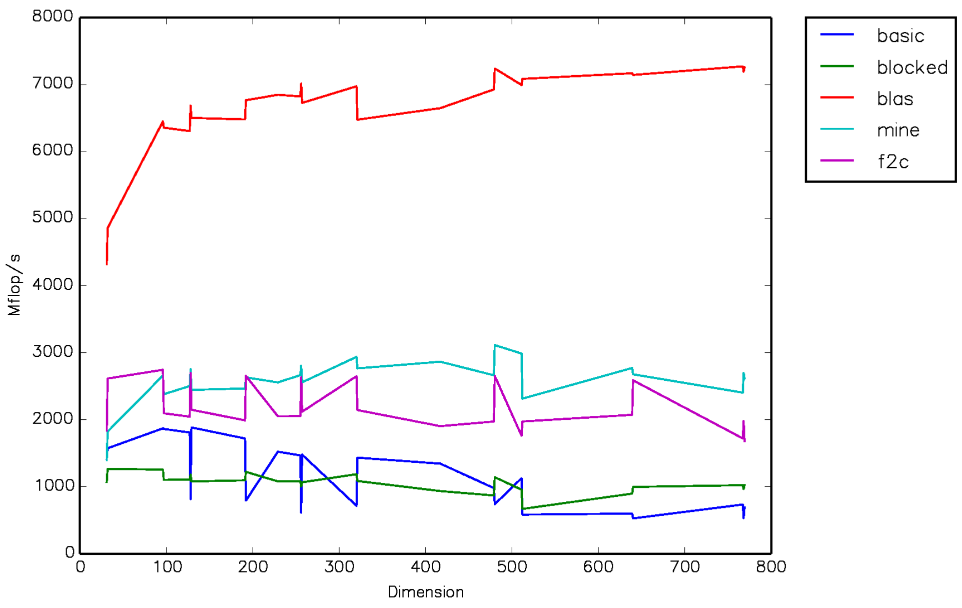

Figure 3: Results adding the -prefetch flag.

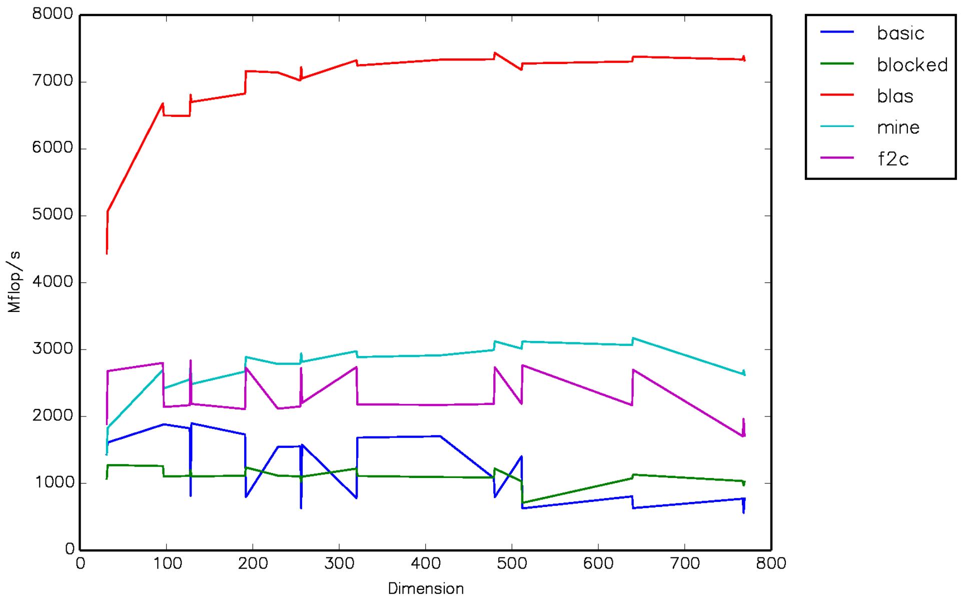

Figure 4: Results adding the -unroll-aggressive flag.