An Attempt at Optimizing Cache Performance: Single-Thread Application

Hirotaka Miura

November 30, 2012

Motivation and Objectives

- Motivation
  - There is constant demand for faster code
  - In single-threaded applications, code can be accelerated by optimizing cache performance which exploits temporal and spatial locality
- Objectives
  - Beginning with a simple code, attempt to minimize the performance gap between manual implementation and Intel Math Kernel Library (MKL).
    - Focus on implementing loop interchange and blocking then applying vectorization

Technical Information

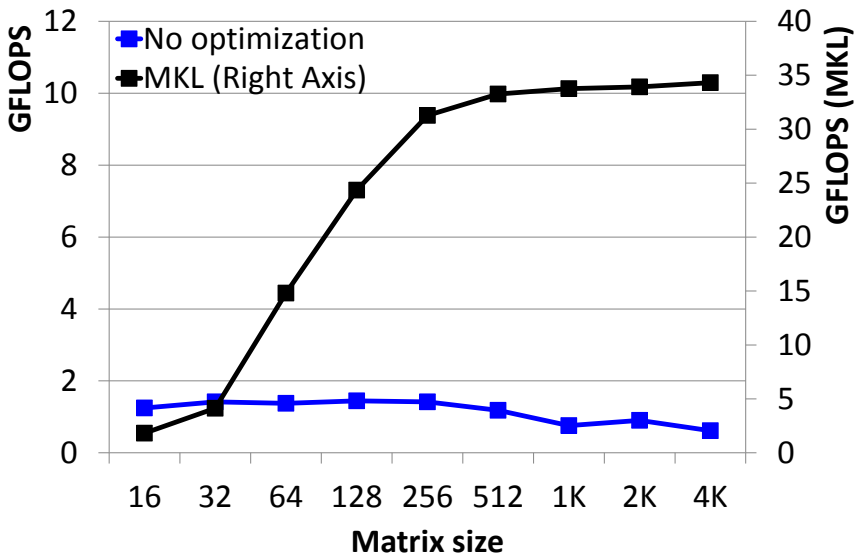
- Machine: lxclab2
- Processor: Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30GHz
- L1 cache size: 32K
  - L1 data cache line size: 64B
- L2 cache size: 256K
- L3 cache size: 15360K
- Performance counter: use Intel Integrated Performance Primitives (IPP) functions.
- Memory allocation: following Hosomi (PR)'sadvice, use memalign() to instruct the program to allocate memory with an offset to avoid cache line conflicts. Memory address is set to be a multiple of 64.

$$lda = N + 128/sizeof(float);$$
$$X = memalign(64, N * sizeof(float) * lda);$$

No Optimization

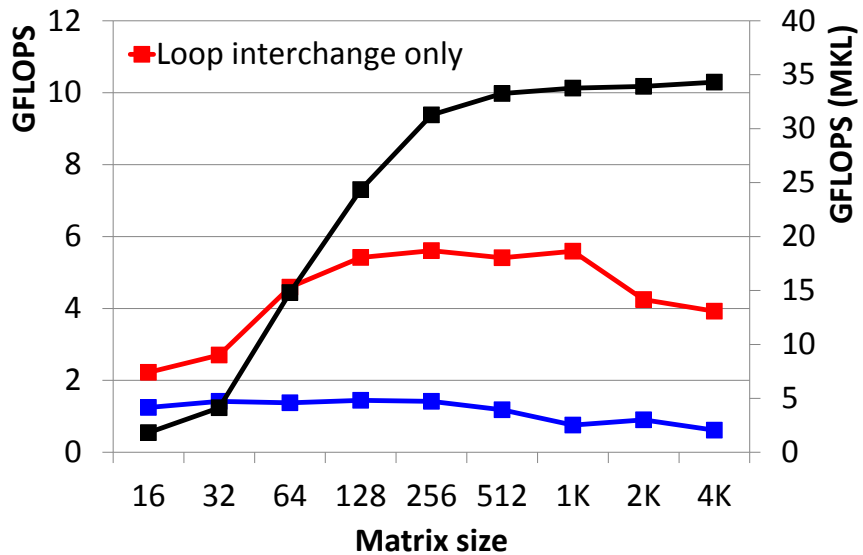
- Poor performance
- Performance degrades as matrix size increases, likely due to increasing cache misses

$$
\begin{aligned}
&for(i = 0; i < N; i++)\\
&\quad for(j = 0; j < N; j++)\\
&\quad\quad for(k = 0; k < N; k++)\\
&\quad\quad\quad C[i * lda + j] += A[i * lda + k] * B[k * lda + j];
\end{aligned}
$$



Loop Interchange

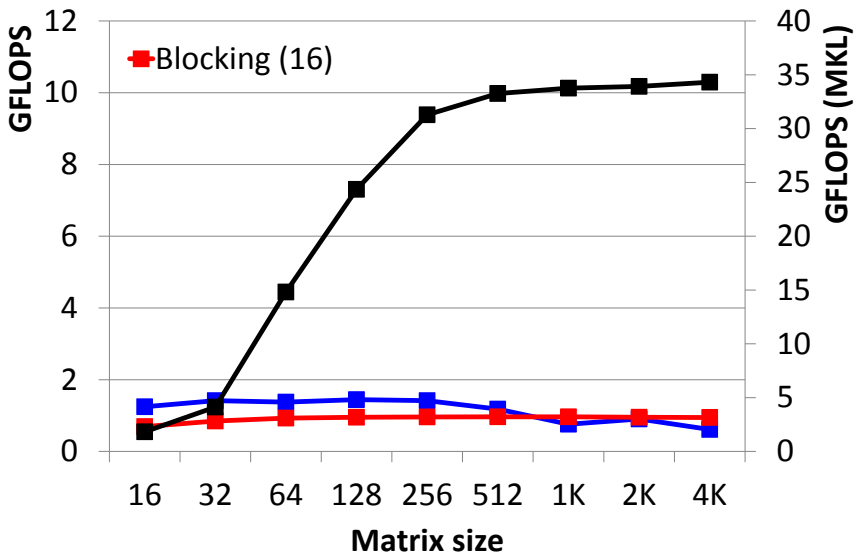
- Temporary increase in performance likely due to exploiting spatial locality of sequential access
- Performance degradation at larger matrices likely due to increased cache misses



$$
\begin{aligned}
&for(i = 0; i < N; i++)\\
&\quad for(k = 0; k < N; k++)\\
&\quad\quad for(j = 0; j < N; j++)\\
&\quad\quad\quad C[i * lda + j] += A[i * lda + k] * B[k * lda + j];
\end{aligned}
$$

Blocking Using Block Size 16

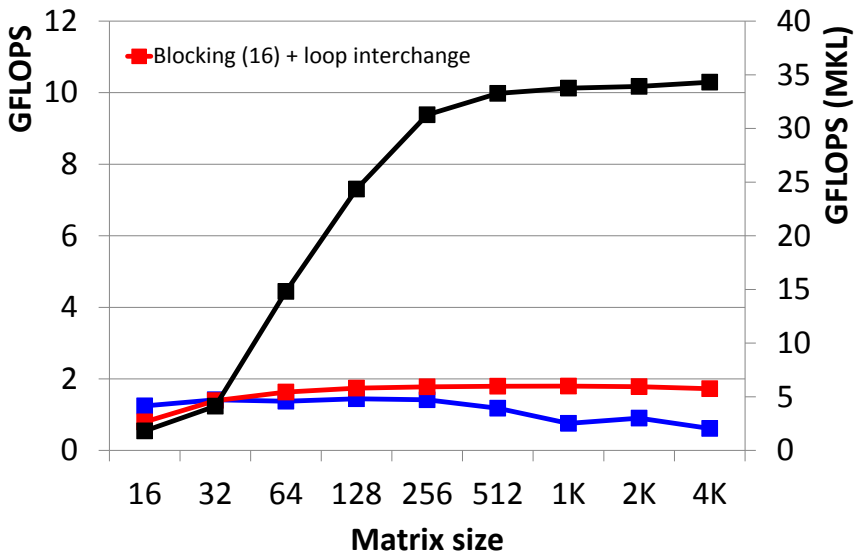
- Blocking seems to reduce performance degradation at larger matrix sizes
- Blocking should help reduce cache misses by improving temporal locality




```
for(ii = 0; ii < N; ii+= BLK){maxii = min(ii + BLK, N);
for(jj = 0; jj < N; jj+= BLK){maxjj = min(jj + BLK, N);
for(kk = 0; kk < N; kk+= BLK){maxkk = min(kk + BLK, N);
for(i = ii; i < maxii; i++)
  for(j = jj; j < maxjj; j++)
    for(k = kk; k < maxkk; k++)
      C[i * lda + j]+= A[i * lda + k] * B[k * lda + j];
}}}
```

Blocking (16) + Loop Interchange

- Blocking stabilizes performance at larger matrices
- Overall performance down considerably compared to using only loop interchange (but almost double the performance of just blocking)
- Block size too small?

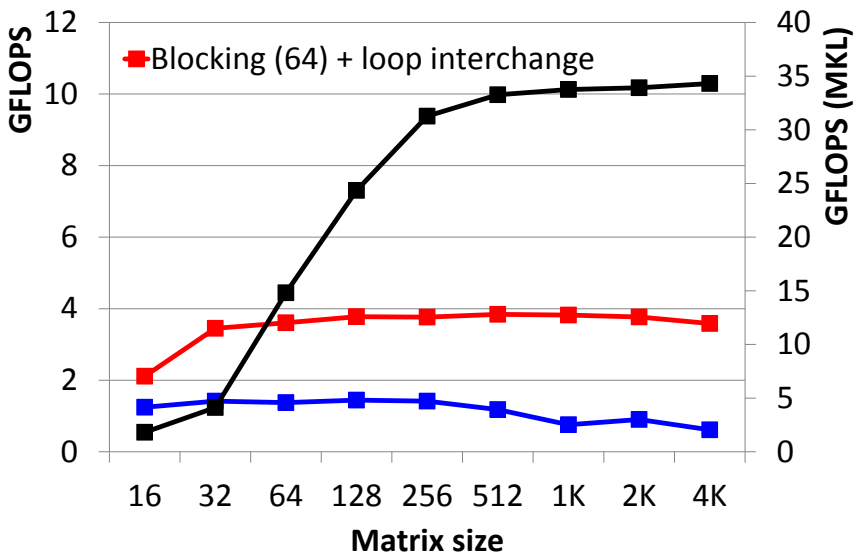



```
for(ii = 0; ii < N; ii+= BLK){maxii = min(ii + BLK, N);
for(kk = 0; kk < N; kk+= BLK){maxkk = min(kk + BLK, N);
for(jj = 0; jj < N; jj+= BLK){maxjj = min(jj + BLK, N);
for(i = ii; i < maxii; i++)
  for(k = kk; k < maxkk; k++)
    for(j = jj; j < maxjj; j++)
      C[i * lda + j]+= A[i * lda + k] * B[k * lda + j];
}}}
```

Blocking (64) + Loop Interchange

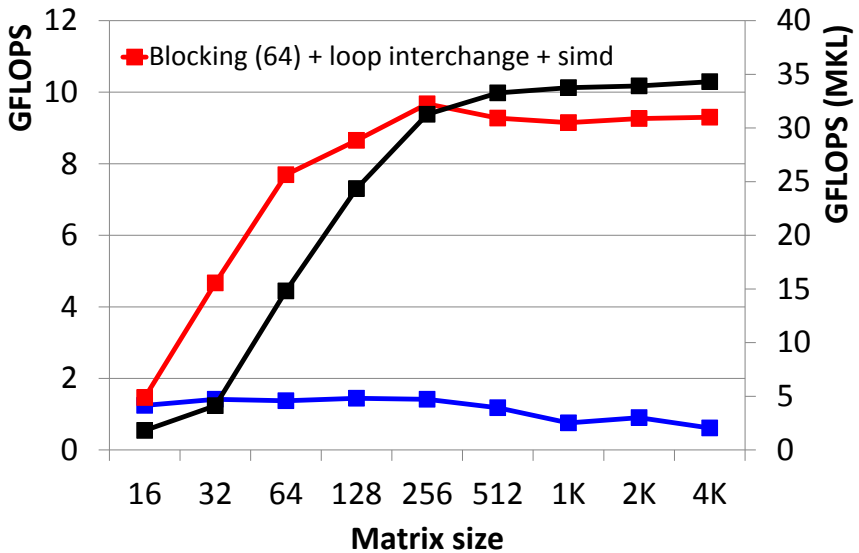
- Increasing block size improves performance, but still slow

```
for(ii = 0; ii < N; ii+= BLK){maxii = min(ii + BLK, N);
for(kk = 0; kk < N; kk+= BLK){maxkk = min(kk + BLK, N);
for(jj = 0; jj < N; jj+= BLK){maxjj = min(jj + BLK, N);
for(i = ii; i < maxii; i++)
  for(k = kk; k < maxkk; k++)
    for(j = jj; j < maxjj; j++)
      C[i * lda + j]+= A[i * lda + k] * B[k * lda + j];
}}}
```



Blocking (64) + Loop Interchange + simd

- Performance jumps, but we fail to achieve SIMD scaling of 4X as in Satish et al. (2012)
- Performance does seem to reach steady state




```
for(ii = 0; ii < N; ii+ = BLK){maxii = min(ii + BLK, N);
for(kk = 0; kk < N; kk+ = BLK){maxkk = min(kk + BLK, N);
for(jj = 0; jj < N; jj+ = BLK){maxjj = min(jj + BLK, N);
for(i = ii; i < maxii; i + +)
  for(k = kk; k < maxkk; k + +)
    #pragma simd
    #pragma vector aligned
    for(j = jj; j < maxjj; j + +)
      C[i * lda + j]+ = A[i * lda + k] * B[k * lda + j];
}}}
```

Conclusion

- We succeed in achieving about 1/4 of MKL performance at steady state levels
- Blocking stabilizes performance at larger matrix sizes and helps in achieving steady stated
- We fail to get expected scaling with the use of simd pragma
- Architectural bottlenecks need to be identified
  - Try using Cilk and elemental functions for outer loop vectorization
  - MKL achieves steady state after 256 x 256 matrix - at the size where L2 cache fills up
    - Need to determine how to more efficiently use L2 cache

References I

Satish, N., C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey (2012): "Can traditional programming bridge the Ninja performance gap for parallel computing applications?," *SIGARCH Comput. Archit. News*, 40(3), 440–451.