5.15 Going faster: Cache blocking and matrix multiply

This section has been set as optional by your instructor.

Our next step in the continuing saga of improving performance of DGEMM by tailoring it to the underlying hardware is to add cache blocking to the subword parallelism and instruction level parallelism optimizations of COD Chapters 3 (Arithmetic for Computers) and 4 (The Processor). The figure below shows the blocked version of DGEMM from COD Figure 4.78 (Optimized C version of DGEMM using C intrinsics ...). The changes are the same as was made earlier in going from unoptimized DGEMM in COD Figure 3.22 (Unoptimized C version of a double precision matrix multiply ...) to blocked DGEMM in COD Figure 5.21 (Cache blocked version of DGEMM). This time we take the unrolled version of DGEMM from COD Chapter 4 (The Processor) and invoke it many times on the submatrices of `A`, `B`, and `C`. Indeed, lines 28—34 and lines 7—8 in the figure below mirror lines 14—20 and lines 5—6 in COD Figure 5.21 (Cache blocked version of DGEMM), except for incrementing the for loop in line 7 by the amount unrolled.

Figure 5.15.1: Optimized C version of DGEMM using cache blocking (COD Figure 5.47).

Optimized C version of DGEMM from COD Figure 4.78 (Optimized C version of DGEMM using C intrins ...).

These changes are the same ones found in COD Figure 5.21 (Cache blocked version of DGEMM). The assembly language produced by the compiler for the `do_block` function is nearly identical to COD Fig 4.79 (The x86 assembly language for the body of the nested loops ...). Once again, there is no overhea call the `do_block` because the compiler inlines the function call.

```
1 #include <x86intrin.h>
2 #define UNROLL (4)
3 #define BLOCKSIZE 32
4 void do_block (int n, int si, int sj, int sk,
5                double *A, double *B, double *C)
6 {
7    for ( int i = si; i < si + BLOCKSIZE; i += UNROLL * 4 )
8       for ( int j = sj; j < sj + BLOCKSIZE; j++ ) {
9          __m256d c[4];
10         for ( int x = 0; x < UNROLL; x++ )
11            c[x] = _mm256_load_pd(C + i + x * 4 + j * n);
12         /* c[x] = C[i][j] */
13         for( int k = sk; k < sk + BLOCKSIZE; k++ )
14         {
15            __m256d b = _mm256_broadcast_sd(B + k + j * n);
16            /*  b = B[k][j] */
17         for (int x = 0; x < UNROLL; x++)
18            c[x] = _mm256_add_pd(c[x], /* c[x] += A[i][k]*b */
19                   _mm256_mul_pd(_mm256_load_pd(A + n * k + x * 4 + i),
20         }
21
22
23         for ( int x = 0; x < UNROLL; x++ )
24            _mm256_store_pd(C + i + x * 4 + j * n, c[x]);
            /* C[i][j] = c[x] */
25      }
26   }
27
28 void dgemm (int n, double* A, double* B, double* C)
29 {
30    for ( int sj = 0; sj < n; sj += BLOCKSIZE )
31       for ( int si = 0; si < n; si += BLOCKSIZE )
32          for ( int sk = 0; sk < n; sk += BLOCKSIZE )
33             do_block(n, si, sj, sk, A, B, C);
34 }
```

Figure 5.15.2: The list of assembly language instructions for the systems and special operations in the full ARMv8 Instruction set. Bold means the Instruction is also in LEGv8 (COD Figure 5.48).

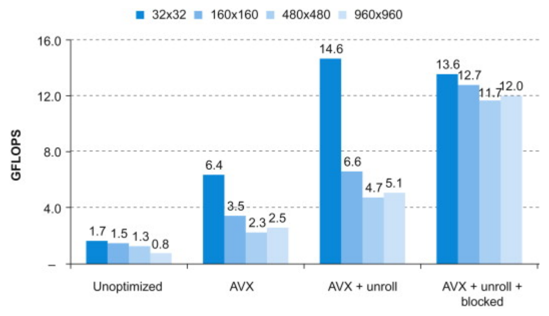
Type	Mnemonic	Instruction
Non-cache	LDNP	Load Non-temporal Pair
	STNP	Store Non-temporal Pair
Barrier	CLREX	Clear exclusive monitor
	DSB	Data synchronization barrier
	DMB	Data memory barrier
	ISB	Instruction synchronization barrier
CRC	CRC32B	CRC-32 sum from byte
	CRC32H	CRC-32 sum from halfword
	CRC32W	CRC-32 sum from word
	CRC32X	CRC-32 sum from doubleword
	CRC32CB	CRC-32C sum from byte
	CRC32CH	CRC-32C sum from halfword
	CRC32CW	CRC-32C sum from word
	CRC32CX	CRC-32C sum from doubleword
Crypto	AESD	AES single round decryption
	AESE	AES single round encryption
	AESIMC	AES inverse mix columns
	AESMC	AES mix columns
	PMULL	Polynomial multiply long
	SHA1C	SHA1 hash update (choose)
	SHA1H	SHA1 fixed rotate
	SHA1M	SHA1 hash update (majority)
	SHA1P	SHA1 hash update (parity)
	SHA1SU0	SHA1 schedule update 0
	SHA1SU1	SHA1 schedule update 1
	SHA256H	SHA256 hash update (part 1)
	SHA256H2	SHA256 hash update (part 2)
	SHA256SU0	SHA256 schedule update 0
	SHA256SU1	SHA256 schedule update 1
Sys Reg	MRS	Move system register to general-purpose register
	MSR	Move general-purpose register or immediate to system register

Type	Mnemonic	Instruction
Unprivileged	LDTR	Load Unprivileged register
	LDTRB	Load Unprivileged byte
	LDTRSB	Load Unprivileged signed byte
	LDTRH	Load Unprivileged halfword
	LDTRSH	Load Unprivileged signed halfword
	LDTRSW	Load Unprivileged signed word
	STTR	Store Unprivileged register
	STTRB	Store Unprivileged byte
	STTRH	Store Unprivileged halfword
Exception	BRK	Software breakpoint instruction
	HLT	Halting software breakpoint instruction
	HVC	Generate exception targeting Exception level 2
	SMC	Generate exception targeting Exception level 3
	SVC	Generate exception targeting Exception level 1
	ERET	Exception return using current ELR and SPSR
Debug	DCPS1	Debug switch to Exception level 1
	DCPS2	Debug switch to Exception level 2
	DCPS3	Debug switch to Exception level 3
	DRPS	Debug restore PE state
System	SYS	System instruction
	SYSL	System instruction with result
	IC	Instruction cache maintenance
	DC	Data cache maintenance
	AT	Address translation
	TLBI	TLB Invalidate
Hint	NOP	No operation
	YIELD	Yield hint
	WFE	Wait for event
	WFI	Wait for interrupt
	SEV	Send event
	SEVL	Send event local
	HINT	Unallocated hint

Unlike the earlier chapters, we do not show the resulting x86 code because the inner loop code is nearly identical to COD Figure 4.79 (The x86 assembly language for the body of the nested loops ...), as the blocking does not affect the computation, just the order that it accesses data in memory. What does change is the bookkeeping integer instructions to implement the loops. It expands from 14 instructions before the inner loop and eight after the loop for COD Figure 4.78 (Optimized C version of DGEMM using C intrinsics ...) to 40 and 28 instructions respectively for the bookkeeping code generated for COD Figure 5.47 (Optimized C version of DGEMM using cache blocking). Nevertheless, the extra instructions executed pale in comparison to the performance improvement of reducing cache misses. The figure below compares unoptimized to optimized for subword parallelism, instruction level parallelism, and caches. Blocking improves performance over unrolled AVX code by factors of 2 to 2.5 for the larger matrices. When we compare unoptimized code to the code with all three optimizations, the performance improvement is factors of 8 to 15, with the largest increase for the largest matrix.

Figure 5.15.3: Performance of four versions of DGEMM from matrix dimensions 32x32 to 960x960 (COD Figure 5.49).

The fully optimized code for the largest matrix is almost 15 times as fast the unoptimized version in COD Figure 3.22 (Unoptimized C version of a double precision matrix multiply ...) in COD Chapter 3 (Arithmetic for Computers).



Elaboration

As mentioned in the Elaboration in COD Section 3.9 (Going faster: Subword parallelism and matrix multiply), these results are with Turbo mode turned off. As in COD Chapters 3 (Arithmetic for Computers) and 4 (The Processor), when we turn it on, we improve all the results by the temporary increase in the clock rate of 3.3/2.6 = 1.27. Turbo mode works particularly well in this case because it is using only a single core of an eight-core chip. However, if we want to run fast we should use all cores, which we'll see in COD Chapter 6 (Parallel Processor from Client to Cloud)

PARTICIPATION ACTIVITY 5.15.1: Cache blocking.

1) A significant amount of overhead is needed to call the `do_block` function in the cache blocked version of DGEMM.

○ True

○ False

2) Cache blocking is used to reduce cache

misses.

- ○ True
- ○ False

Provide feedback on this section