To support algorithms requiring unaligned 128-bit SIMD memory accesses, memory buffer allocation by a caller function should consider adding some pad space so that a callee function can safely use the address pointer safely with unaligned 128-bit SIMD memory operations.

The minimal padding size should be the width of the SIMD register that might be used in conjunction with unaligned SIMD memory access.

10.2.2 Unaligned Memory Access and String Library

String library functions may be used by application code or privileged code. String library functions must be careful not to violate memory access rights. Therefore, a replacement string library that employ SIMD unaligned access must employ special techniques to ensure no memory access violation occur.

Section 10.3.6 provides an example of a replacement string library function implemented with SSE4.2 and demonstrates a technique to use 128-bit unaligned memory access without unintentionally crossing page boundary.

10.3 SSE4.2 APPLICATION CODING GUIDELINE AND EXAMPLES

Software implementing SSE4.2 instruction must use CPUID feature flag mechanism to verify processor's support for SSE4.2. Details can be found in CHAPTER 12 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* and in CPUID of CHAPTER 3 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

In the following sections, we use several examples in string/text processing of progressive complexity to illustrates the basic techniques of adapting the SIMD approach to implement string/text processing using PCMPxSTRy instructions in SSE4.2. For simplicity, we will consider string/text in byte data format in situations that caller functions have allocated sufficient buffer size to support unaligned 128-bit SIMD loads from memory without encountering side-effects of cross page boundaries.

10.3.1 Null Character Identification (Strlen equivalent)

The most widely used string function is probably strlen(). One can view the lexing requirement of strlen() is to identify the null character in a text block of unknown size (end of string condition). Brute-force, byte-granular implementation fetches data inefficiently by loading one byte at a time.

Optimized implementation using general-purpose instructions can take advantage of dword operations in 32-bit environment (and qword operations in 64-bit environment) to reduce the number of iterations.

A 32-bit assembly implementation of strlen() is shown Example 10-3. The peak execution throughput of handling EOS condition is determined by eight ALU instructions in the main loop.

Example 10-3. Strlen() Using General-Purpose Instructions

```
int strlen_asm(const  char* s1)
{int len = 0;
   _asm{
   mov   ecx, s1
   test   ecx, 3 ; test addr aligned to dword
   je      short _main_loop1  ; dword aligned loads would be faster
_malign_str1:
   mov   al, byte ptr [ecx] ; read one byte at a time
   add   ecx, 1
   test   al, al ; if we find a null, go calculate the length
   je      short _byte3a
                (continue)
```

Example 10-3. Strlen() Using General-Purpose Instructions

```
   test   ecx, 3; test if addr is now aligned to dword
   jne    short _malign_str1; if not, repeat
   align16
_main_loop1:; read each 4-byte block and check for a NULL char in the dword
   mov  eax, [ecx]; read 4 byte to reduce loop count
   mov  edx, 7efefeffh
   add   edx, eax
   xor   eax, -1
   xor   eax, edx
   add   ecx, 4; increment address pointer by 4
   test  eax, 81010100h ; if no null code in 4-byte stream, do the next 4 bytes
   je     short _main_loop1
   ;      there is a null char in the dword we just read,
   ;      since we already advanced pointer ecx by 4, and the dword is lost
   mov  eax, [ecx -4]; re-read the dword that contain at least a null char
   test  al, al      ; if byte0 is null
   je     short _byte0a; the least significant byte is null
   test  ah, ah    ; if byte1 is null
   je     short _byte1a
   test  eax, 00ff0000h; if byte2 is null
   je     short _byte2a
   test  eax, 00ff000000h; if byte3 is null
   je     short _byte3a
   jmp   short _main_loop1
_byte3a:
   ; we already found the null, but pointer already advanced by 1
   lea    eax, [ecx-1]; load effective address corresponding to null code
   mov  ecx, s1
   sub   eax, ecx; difference between null code and start address
   jmp   short _resulta
_byte2a:
```

```
   lea    eax, [ecx-2]
   mov  ecx, s1
   sub   eax, ecx
   jmp   short _resulta
_byte1a:
   lea    eax, [ecx-3]
   mov  ecx, s1
   sub   eax, ecx
   jmp   short _resulta
_byte0a:
   lea    eax, [ecx-4]
   mov  ecx, s1
   sub   eax, ecx
_resulta:
   mov    len, eax; store result
   }
   return len;
}
```

The equivalent functionality of EOS identification can be implemented using PCMPISTRI. Example 10-4 shows a simplistic SSE4.2 implementation to scan a text block by loading 16-byte text fragments and locate the null termination character. Example 10-5 shows the optimized SSE4.2 implementation that demonstrates the importance of using memory disambiguation to improve instruction-level parallelism.

Example 10-4. Sub-optimal PCMPISTRI Implementation of EOS handling

```
static char ssch2[16]= {0x1, 0xff, 0x00, }; // range values for non-null characters

int strlen_un_optimized(const  char* s1)
{int len = 0;
  _asm{
  mov  eax, s1
  movdquxmm2, ssch2 ; load character pair as range (0x01 to 0xff)
  xor   ecx, ecx ; initial offset to 0
              (continue)
```

```
_loopc:
  add   eax, ecx   ; update addr pointer to start of text fragment
  pcmpistri  xmm2, [eax], 14h; unsigned bytes, ranges, invert, lsb index returned to ecx
      ; if there is a null char in the 16Byte fragment at [eax], zf will be set.
      ; if all 16 bytes of the fragment are non-null characters, ECX will return 16,
  jnz    short _loopc; xmm1 has no null code, ecx has 16, continue search
  ;       we have a null code in xmm1, ecx has the offset of the null code i
  add   eax, ecx       ; add ecx to the address of the last fragment2/xmm1
  mov  edx, s1; retrieve effective address of the input string
  sub   eax, edx;the string length
  mov  len, eax; store result
  }
  return len;
}
```

The code sequence shown in Example 10-4 has a loop consisting of three instructions. From a performance tuning perspective, the loop iteration has loop-carry dependency because address update is done using the result (ECX value) of a previous loop iteration. This loop-carry dependency deprives the out-of-order engine's capability to have multiple iterations of the instruction sequence making forward progress. The latency of memory loads, the latency of these instructions, any bypass delay could not be amortized by OOO execution in the presence of loop-carry dependency.

A simple optimization technique to eliminate loop-carry dependency is shown in Example 10-5.

Using memory disambiguation technique to eliminate loop-carry dependency, the cumulative latency exposure of the 3-instruction sequence of Example 10-5 is amortized over multiple iterations, the net cost of executing each iteration (handling 16 bytes) is less then 3 cycles. In contrast, handling 4 bytes of string data using 8 ALU instructions in Example 10-3 will also take a little less than 3 cycles per iteration. Whereas each iteration of the code sequence in Example 10-4 will take more than 10 cycles because of loop-carry dependency.

Example 10-5. Strlen() Using PCMPISTRI without Loop-Carry Dependency

```
int strlen_sse4_2(const  char* s1)
{int len = 0;
  _asm{
  mov  eax, s1
  movdquxmm2, ssch2 ; load character pair as range (0x01 to 0xff)
  xor   ecx, ecx ; initial offset to 0
  sub   eax, 16 ; address arithmetic to eliminate extra instruction and a branch
```

Example 10-5. Strlen() Using PCMPISTRI without Loop-Carry Dependency

```
_loopc:
   add    eax, 16 ; adjust address pointer and disambiguate load address for each iteration
   pcmpistri  xmm2, [eax], 14h; unsigned bytes, ranges, invert, lsb index returned to ecx
       ; if there is a null char in [eax] fragment, zf will be set.
       ; if all 16 bytes of the fragment are non-null characters, ECX will return 16,
   jnz short _loopc ; ECX will be 16 if there is no null byte in [eax], so we disambiguate
_endofstring:
   add    eax, ecx       ; add ecx to the address of the last fragment
   mov   edx, s1; retrieve effective address of the input string
   sub    eax, edx;the string length
   mov   len, eax; store result
   }
   return len;
}
```

SSE4.2 Coding Rule 5. (H impact, H generality) *Loop-carry dependency that depends on the ECX result of PCMPESTRI/PCMPESTRM/PCMPISTRI/PCMPISTRM for address adjustment must be minimized. Isolate code paths that expect ECX result will be 16 (bytes) or 8 (words), replace these values of ECX with constants in address adjustment expressions to take advantage of memory disambiguation hardware.*

10.3.2 White-Space-Like Character Identification

Character-granular-based text processing algorithms have developed techniques to handle specific tasks to remedy the efficiency issue of character-granular approaches. One such technique is using look-up tables for character subset classification. For example, some application may need to separate alpha-numeric characters from white-space-like characters. More than one character may be treated as white-space characters.

Example 10-6 illustrates a simple situation of identifying white-space-like characters for the purpose of marking the beginning and end of consecutive non-white-space characters.

Example 10-6. WordCnt() Using C and Byte-Scanning Technique

```
// Counting words involves locating the boundary of contiguous non-whitespace characters.
// Different software may choose its own mapping of white space character set.
// This example employs a simple definition for tutorial purpose:
// Non-whitespace character set will consider: A-Z, a-z, 0-9, and the apostrophe mark '
// The example uses a simple technique to map characters into bit patterns of square waves
// we can simply count the number of falling edges

static char alphnrange[16]= {0x27, 0x27, 0x30, 0x39, 0x41, 0x5a, 0x61, 0x7a, 0x0};
static char alp_map8[32] = {0x0, 0x0, 0x0, 0x0, 0x80,0x0,0xff, 0x3,0xfe, 0xff, 0xff, 0x7, 0xfe, 0xff, 0xff, 0x7}; // 32
byte lookup table, 1s map to bit patterns of alpha numerics in alphnrange
int wordcnt_c(const  char* s1)
{int  i, j, cnt = 0;
char cc, cc2;
char  flg[3]; // capture the a wavelet to locate a falling edge
   cc2 = cc = s1[0];
   // use the compacted bit pattern to consolidate multiple comparisons into one look up
   if( alp_map8[cc>>3] & ( 1<< ( cc & 7) ) )
   {    flg[1] = 1; } // non-white-space char that is part of a word,
                (continue)
```

Example 10-6. WordCnt() Using C and Byte-Scanning Technique

```
  // we're including apostrophe in this example since counting the
  // following 's' as a separate word would be kind of silly
  else
  {   flg[1] = 0; }  // 0: whitespace, punctuations not be considered as part of a word

  i = 1; // now we're ready to scan through the rest of the block
  // we'll try to pick out each falling edge of the bit pattern to increment word count.
  // this works with consecutive white spaces, dealing with punctuation marks, and
  // treating hyphens as connecting two separate words.
  while (cc2 )
  {   cc2 = s1[i];
      if( alp_map8[cc2>>3] & ( 1<< ( cc2 & 7) ) )
      {   flg[2] = 1;} // non-white-space
      else
      {   flg[2] = 0;} // white-space-like

      if( !flg[2] && flg[1] )
      {   cnt ++; }// found the falling edge
      flg[1] = flg[2];
      i++;
  }
  return cnt;
}
```

In Example 10-6, a 32-byte look-up table is constructed to represent the ascii code values 0x0-0xff, and partitioned with each bit of 1 corresponding to the specified subset of characters. While this bit-lookup technique simplifies the comparison operations, data fetching remains byte-granular.

Example 10-7 shows an equivalent implementation of counting words using PCMPISTRM. The loop iteration is performed at 16-byte granularity instead of byte granularity. Additionally, character set subsetting is easily expressed using range value pairs and parallel comparisons between the range values and each byte in the text fragment are performed by executing PCMPISTRI once.

Example 10-7. WordCnt() Using PCMPISTRM

```
// an SSE4.2 example of counting words using the definition of non-whitespace character
// set of {A-Z, a-z, 0-9, '}. Each text fragment (up to 16 bytes) are mapped to a
// 16-bit pattern, which may contain one or more falling edges. Scanning bit-by-bit
// would be inefficient and goes counter to leveraging SIMD programming techniques.
// Since each falling edge must have a preceding rising edge, we take a finite
// difference approach to derive a pattern where each rising/falling edge maps to 2-bit pulse,
// count the number of bits in the 2-bit pulses using popcnt and divide by two.
int wdcnt_sse4_2(const  char* s1)
{int len = 0;
  _asm{
  mov  eax, s1
  movdquxmm3, alphnrange ; load range value pairs to detect non-white-space codes
  xor   ecx, ecx
  xor   esi, esi
  xor   edx, edx
              (continue)
```

Example 10-7. WordCnt() Using PCMPISTRM

```
    movdquxmm1, [eax]
    pcmpistrm  xmm3, xmm1, 04h ; white-space-like char becomes 0 in xmm0[15:0]
    movdqa xmm4, xmm0
    movdqa     xmm1, xmm0
    psrld xmm4, 15    ; save MSB to use in next iteration
    movdqa xmm5, xmm1
    psllw xmm5, 1; lsb is effectively mapped to a white space
    pxor  xmm5, xmm0; the first edge is due to the artifact above
    pextrd edi, xmm5, 0
    jz      _lastfragment; if xmm1 had a null, zf would be set
    popcnt edi, edi; the first fragment will include a rising edge
    add      esi, edi
    mov  ecx, 16
_loopc:
    add   eax, ecx ; advance address pointer
    movdquxmm1, [eax]
    pcmpistrm  xmm3, xmm1, 04h ; white-space-like char becomes 0 in xmm0[15:0]
    movdqa xmm5, xmm4  ; retrieve the MSB of the mask from last iteration
    movdqa xmm4, xmm0
    psrld xmm4, 15    ; save mSB of this iteration for use in next iteration
    movdqa xmm1, xmm0
```

```
    psllw xmm1, 1
    por    xmm5, xmm1  ; combine MSB of last iter and the rest from current iter
    pxor   xmm5, xmm0; differentiate binary wave form into pattern of edges
    pextrdedi, xmm5, 0 ; the edge patterns has (1 bit from last, 15 bits from this round)
    jz      _lastfragment; if xmm1 had a null, zf would be set
    mov  ecx, 16; xmm1, had no null char, advance 16 bytes
    popcntedi, edi; count both rising and trailing edges
    add   esi, edi; keep a running count of both edges
    jmp   short _loopc
_lastfragment:
    popcntedi, edi; count both rising and trailing edges
    add   esi, edi; keep a running count of both edges
    shr    esi, 1 ; word count corresponds to the trailing edges
    mov  len, esi
    }
    return len;
}
```

10.3.3 Substring Searches

Strstr() is a common function in the standard string library. Typically, A library may implement strstr(sTarg, sRef) with a brute-force, byte-granular technique of iterative comparisons between the reference string with a round of string comparison with a subset of the target string. Brute-force, byte-granular techniques provide reasonable efficiency when the first character of the target substring and the reference string are different, allowing subsequent string comparisons of target substrings to proceed forward to the next byte in the target string.

When a string comparison encounters partial matches of several characters (i.e. the sub-string search found a partial match starting from the beginning of the reference string) and determined the partial match led to a false-match. The brute-force search process need to go backward and restart string comparisons from a location that had participated in previous string comparison operations. This is referred to as re-trace inefficiency of the brute-force substring search algorithm. See Figure 10-2.

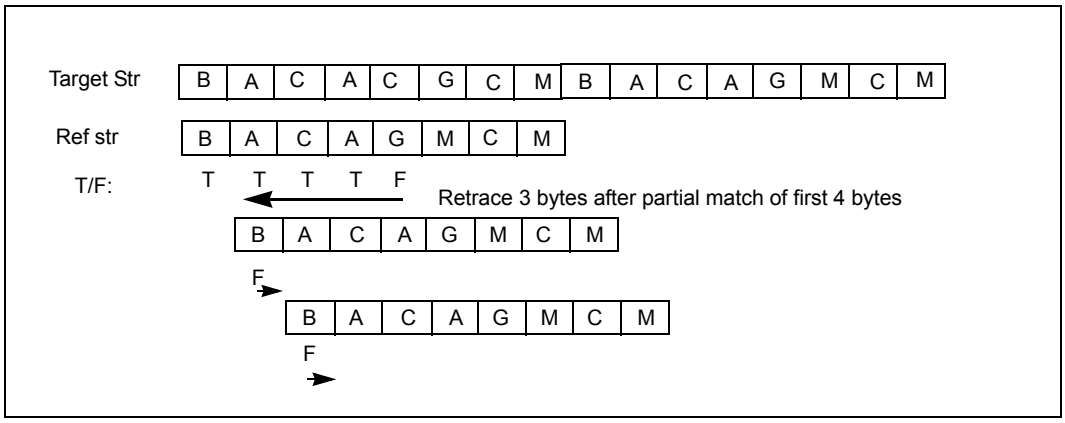

Figure 10-2. Retrace Inefficiency of Byte-Granular, Brute-Force Search

The Knuth, Morris, Pratt algorithm[1] (KMP) provides an elegant enhancement to overcome the re-trace inefficiency of brute-force substring searches. By deriving an overlap table that is used to manage retrace distance when a partial match leads to a false match, KMP algorithm is very useful for applications that search relevant articles containing keywords from a large corpus of documents.

Example 10-8 illustrates a C-code example of using KMP substring searches.

Example 10-8. KMP Substring Search in C

```
// s1 is the target string of length cnt1
// s2 is the reference string of length cnt2
// j is the offset in target string s1 to start each round of string comparison
// i is the offset in reference string s2 to perform byte granular comparison
                (continue)
```

```
int str_kmp_c(const  char* s1, int cnt1, const  char* s2, int cnt2 )
{ int i, j;
    i = 0; j = 0;
    while ( i+j < cnt1) {
        if( s2[i] == s1[i+j]) {
            i++;
            if( i == cnt2) break; // found full match
        }
        else {
            j = j+i - ovrlap_tbl[i]; // update the offset in s1 to start next round of string compare
            if( i > 0) {
                i = ovrlap_tbl[i]; // update the offset of s2 for next string compare should start at
            }
        }
    };
 return j;
}
```

1. Donald E. Knuth, James H. Morris, and Vaughan R. Pratt; SIAM J. Comput. Volume 6, Issue 2, pp. 323-350 (1977)

Example 10-8. KMP Substring Search in C

```
void kmp_precalc(const char * s2, int cnt2)
{int i = 2;
char nch = 0;
   ovrlap_tbl[0] = -1; ovrlap_tbl[1] = 0;
      // pre-calculate KMP table
   while( i < cnt2) {
      if( s2[i-1] == s2[nch]) {
         ovrlap_tbl[i] =  nch +1;
         i++; nch++;
      }
      else if ( nch > 0) nch = ovrlap_tbl[nch];
      else {
         ovrlap_tbl[i] = 0;
         i++;
      }
   };
   ovrlap_tbl[cnt2] = 0;
}
```

Example 10-8 also includes the calculation of the KMP overlap table. Typical usage of KMP algorithm involves multiple invocation of the same reference string, so the overhead of precalculating the overlap table is easily amortized. When a false match is determined at offset i of the reference string, the overlap table will predict where the next round of string comparison should start (updating the offset j), and the offset in the reference string that byte-granular character comparison should resume/restart.

While KMP algorithm provides efficiency improvement over brute-force byte-granular substring search, its best performance is still limited by the number of byte-granular operations. To demonstrate the versatility and built-in lexical capability of PCMPISTRI, we show an SSE4.2 implementation of substring search using brute-force 16-byte granular approach in Example 10-9, and combining KMP overlap table with substring search using PCMPISTRI in Example 10-10.

Example 10-9. Brute-Force Substring Search Using PCMPISTRI Intrinsic

```
int strsubs_sse4_2i(const  char* s1, int cnt1, const  char* s2, int cnt2 )
{ int  kpm_i=0, idx;
int  ln1= 16, ln2=16, rcnt1 = cnt1, rcnt2= cnt2;
__m128i *p1 = (__m128i *) s1;
__m128i *p2 = (__m128i *) s2;
__m128ifrag1, frag2;
int  cmp, cmp2,  cmp_s;
__m128i *pt = NULL;
    if( cnt2 > cnt1 || !cnt1) return -1;
    frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
    frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment

                (continue)
```

Example 10-9. Brute-Force Substring Search Using PCMPISTRI Intrinsic

```
while(rcnt1 > 0)
{   cmp_s = _mm_cmpestrs(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x0c);
    cmp = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x0c);
    if( !cmp) { // we have a partial match that needs further analysis
        if( cmp_s) { // if we're done with s2
            if( pt)
            {idx = (int) ((char *) pt - (char *) s1) ; }
            else
            {idx = (int) ((char *) p1 - (char *) s1) ;  }
            return idx;
        }
        // we do a round of string compare to verify full match till end of s2
        if( pt == NULL) pt = p1;
        cmp2 = 16;
        rcnt2 = cnt2  - 16 -(int) ((char *)p2-(char *)s2);
        while(  cmp2 == 16  && rcnt2) {  // each 16B frag matches,
            rcnt1 = cnt1  - 16 -(int) ((char *)p1-(char *)s1);
            rcnt2 = cnt2  - 16 -(int) ((char *)p2-(char *)s2);
            if( rcnt1 <=0 || rcnt2 <= 0 ) break;
            p1 = (__m128i *)(((char *)p1) + 16);
            p2 = (__m128i *)(((char *)p2) + 16);
            frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
            frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
            cmp2 = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x18); // lsb, eq each
        };
        if( !rcnt2 || rcnt2 == cmp2) {
            idx = (int) ((char *) pt - (char *) s1) ;
            return idx;
        }
        else if ( rcnt1 <= 0) { // also cmp2 < 16, non match
            if( cmp2 == 16 && ((rcnt1 + 16) >= (rcnt2+16) ) )
            {idx = (int) ((char *) pt - (char *) s1) ;
            return idx;
            }
            else return -1;
        }
            (continue)
```

Example 10-9. Brute-Force Substring Search Using PCMPISTRI Intrinsic

```
        else { // in brute force, we advance fragment offset in target string s1 by 1
            p1 = (__m128i *)(((char *)pt) + 1);  // we're  not taking advantage of kmp
            rcnt1 = cnt1  -(int) ((char *)p1-(char *)s1);
            pt = NULL;
            p2 = (__m128i *)((char *)s2) ;
            rcnt2 = cnt2  -(int) ((char *)p2-(char *)s2);
            frag1 = _mm_loadu_si128(p1);// load next fragment from s1
            frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
        }
      }
      else{
        if( cmp == 16) p1 = (__m128i *)(((char *)p1) + 16);
        else p1 = (__m128i *)(((char *)p1) + cmp);
        rcnt1 = cnt1  -(int) ((char *)p1-(char *)s1);
        if( pt && cmp ) pt = NULL;
        frag1 = _mm_loadu_si128(p1);// load next fragment from s1
      }
   }
   return idx;
}
```

In Example 10-9, address adjustment using a constant to minimize loop-carry dependency is practised in two places:

- In the inner while loop of string comparison to determine full match or false match (the result cmp2 is not used for address adjustment to avoid dependency).
- In the last code block when the outer loop executed PCMPISTRI to compare 16 sets of ordered compare between a target fragment with the first 16-byte fragment of the reference string, and all 16 ordered compare operations produced false result (producing cmp with a value of 16).

Example 10-10 shows an equivalent intrinsic implementation of substring search using SSE4.2 and KMP overlap table. When the inner loop of string comparison determines a false match, the KMP overlap table is consulted to determine the address offset for the target string fragment and the reference string fragment to minimize retrace.

It should be noted that a significant portions of retrace with retrace distance less than 15 bytes are avoided even in the brute-force SSE4.2 implementation of Example 10-9. This is due to the order-compare primitive of PCMPISTRI. "Ordered compare" performs 16 sets of string fragment compare, and many false match with less than 15 bytes of partial matches can be filtered out in the same iteration that executed PCMPISTRI.

Retrace distance of greater than 15 bytes does not get filtered out by the Example 10-9. By consulting with the KMP overlap table, Example 10-10 can eliminate retraces of greater than 15 bytes.

Example 10-10. Substring Search Using PCMPISTRI and KMP Overlap Table

```
int strkmp_sse4_2(const  char* s1, int cnt1, const  char* s2, int cnt2 )
{ int  kpm_i=0, idx;
int  ln1= 16, ln2=16, rcnt1 = cnt1, rcnt2= cnt2;
__m128i *p1 = (__m128i *) s1;
__m128i *p2 = (__m128i *) s2;
__m128ifrag1, frag2;
                (continue)
```

Example 10-10. Substring Search Using PCMPISTRI and KMP Overlap Table

```
int  cmp, cmp2,  cmp_s;
__m128i *pt = NULL;
    if( cnt2 > cnt1 || !cnt1) return -1;
    frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
    frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment

    while(rcnt1 > 0)
    {   cmp_s = _mm_cmpestrs(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x0c);
        cmp = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x0c);
        if( !cmp) { // we have a partial match that needs further analysis
            if( cmp_s) { // if we've reached the end with s2
                if( pt)
                {idx = (int) ((char *) pt - (char *) s1) ; }
                else
                {idx = (int) ((char *) p1 - (char *) s1) ;  }
                return idx;
            }
        // we do a round of string compare to verify full match till end of s2
            if( pt == NULL) pt = p1;
            cmp2 = 16;
            rcnt2 = cnt2  - 16 -(int) ((char *)p2-(char *)s2);
            while(  cmp2 == 16 && rcnt2) {  // each 16B frag matches
                rcnt1 = cnt1  - 16 -(int) ((char *)p1-(char *)s1);
                rcnt2 = cnt2  - 16 -(int) ((char *)p2-(char *)s2);
                if( rcnt1 <=0 || rcnt2 <= 0 ) break;
                p1 = (__m128i *)(((char *)p1) + 16);
                p2 = (__m128i *)(((char *)p2) + 16);
                frag1 = _mm_loadu_si128(p1);// load up to 16 bytes of fragment
                frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
                cmp2 = _mm_cmpestri(frag2, (rcnt2>ln2)? ln2: rcnt2, frag1, (rcnt1>ln1)? ln1: rcnt1, 0x18); // lsb, eq each
            };
            if( !rcnt2 || rcnt2 == cmp2) {
                idx = (int) ((char *) pt - (char *) s1) ;
                return idx;
            }
            else if ( rcnt1 <= 0 ) { // also cmp2 < 16, non match
                return -1;
            }
                (continue)
```

Example 10-10. Substring Search Using PCMPISTRI and KMP Overlap Table

```
            else { // a partial match led to false match, consult KMP overlap table for addr adjustment
                kpm_i = (int) ((char *)p1 - (char *)pt)+ cmp2 ;
                p1 = (__m128i *)(((char *)pt) + (kpm_i - ovrlap_tbl[kpm_i]));  // use kmp to skip retrace
                rcnt1 = cnt1  -(int) ((char *)p1-(char *)s1);
                pt = NULL;
                p2 = (__m128i *)(((char *)s2) + (ovrlap_tbl[kpm_i]));
                rcnt2 = cnt2  -(int) ((char *)p2-(char *)s2);
                frag1 = _mm_loadu_si128(p1);// load next fragment from s1
                frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
            }
        }
```

```
        else{
                if( kpm_i && ovrlap_tbl[kpm_i]) {
                p2 = (__m128i *)(((char *)s2) );
                frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
                //p1 = (__m128i *)(((char *)p1) );

                //rcnt1 = cnt1  -(int) ((char *)p1-(char *)s1);
                if( pt && cmp ) pt = NULL;
                rcnt2 = cnt2  ;
                //frag1 = _mm_loadu_si128(p1);// load next fragment from s1
                frag2 = _mm_loadu_si128(p2);// load up to 16 bytes of fragment
                kpm_i = 0;
                }
                else { // equ order comp resulted in sub-frag match or non-match
                if( cmp == 16) p1 = (__m128i *)(((char *)p1) + 16);
                else p1 = (__m128i *)(((char *)p1) + cmp);
                rcnt1 = cnt1  -(int) ((char *)p1-(char *)s1);
                if( pt && cmp ) pt = NULL;
                frag1 = _mm_loadu_si128(p1);// load next fragment from s1
                }
        }
    }
    return idx;
}
```

The relative speed up of byte-granular KMP, brute-force SSE4.2, and SSE4.2 with KMP overlap table over byte-granular brute-force substring search is illustrated in the graph that plots relative speedup over percentage of retrace for a reference string of 55 bytes long. A retrace of 40% in the graph meant, after a partial match of the first 22 characters, a false match is determined.

So when brute-force, byte-granular code has to retrace, the other three implementation may be able to avoid the need to retrace because:

- Example 10-8 can use KMP overlap table to predict the start offset of next round of string compare operation after a partial-match/false-match, but forward movement after a first-character-false-match is still byte-granular.

- Example 10-9 can avoid retrace of shorter than 15 bytes but will be subject to retrace of 21 bytes after a partial-match/false-match at byte 22 of the reference string. Forward movement after each order-compare-false-match is 16 byte granular.
- Example 10-10 avoids retrace of 21 bytes after a partial-match/false-match, but KMP overlap table lookup incurs some overhead. Forward movement after each order-compare-false-match is 16 byte granular.

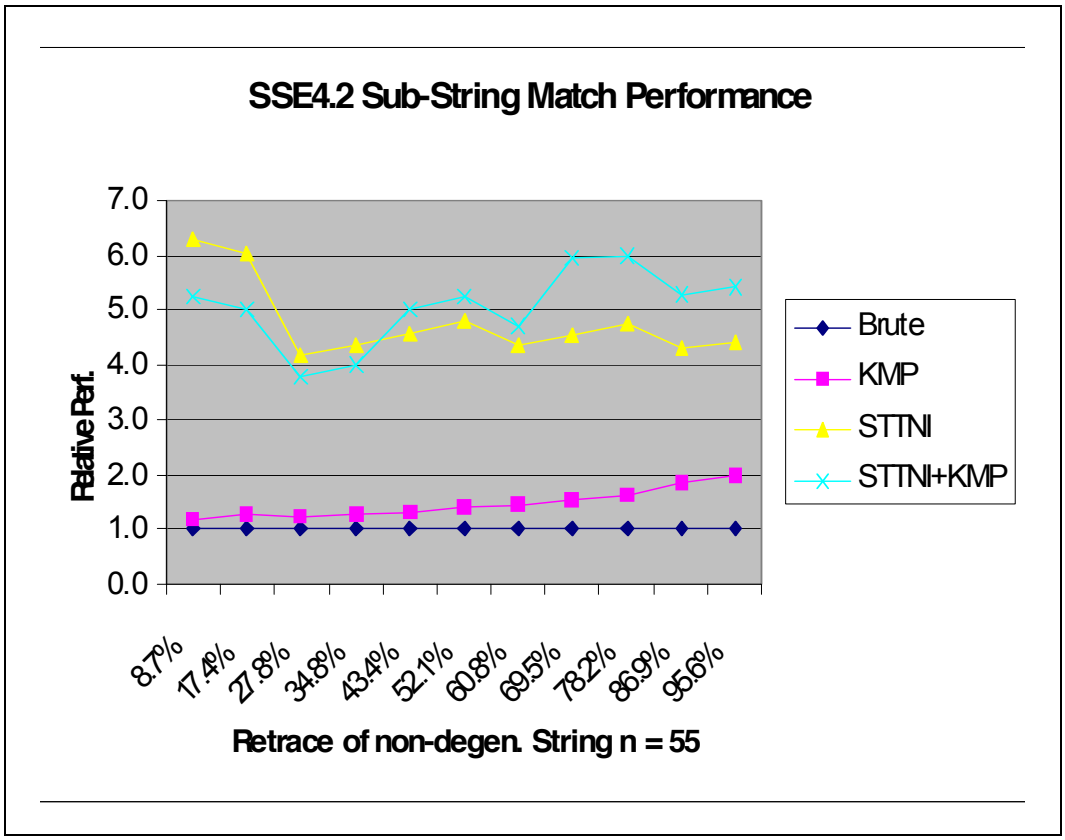

Figure 10-3. SSE4.2 Speedup of SubString Searches

10.3.4 String Token Extraction and Case Handling

Token extraction is a common task in text/string handling. It is one of the foundation of implementing lexer/parser objects of higher sophistication. Indexing services also build on tokenization primitives to sort text data from streams.

Tokenization requires the flexibility to use an array of delimiter characters.

A library implementation of Strtok_s() may employ a table-lookup technique to consolidate sequential comparisons of the delimiter characters into one comparison (similar to Example 10-6). An SSE4.2 implementation of the equivalent functionality of strtok_s() using intrinsic is shown in Example 10-11.