Compiling for "Nehalem"
(the Intel® Core™ Microarchitecture, Intel® Xeon® 5500 processor family and the Intel® Core™ i7 processor)

Martyn Corden

Developer Products Division
Software & Services Group
Intel Corporation

Optimization Guidelines For Intel® Core™ i7 Processor

- Many new features introduced that you get for free
  - Better branch prediction + faster mispredict correction
  - Improvements on unaligned loads + cache-line splits
  - Improvements on store forwarding
  - Memory bandwidth increase
  - Reduced memory latency
  - Etc…
- No large differences in tuning guidelines
  - Still use Intel® 64 and IA-32 Architectures Optimization Reference Manual: http://www.intel.com/products/processor/manuals/
- This presentation will discuss optimizations/recommendations to further enhance performance on Intel® Core™ i7 processor

Streaming SIMD Extensions 4.2 + ATA.1

(SSE4 Efficient Accelerated String and Text Processing instructions)

7 new instructions

- QWORD comparison (1) — image processing
  - PCMPGTQ generated automatically in 11.0
- Byte/Word text processing (4) — string operations
  - used in intrinsics in 11.1
- Accumulation of CRC32 value (1) — cryptography
- Bit counting/popcnt (1) — "

- No new data types
- use 128-bit operand similar to SSE4.1

Streaming SIMD Extensions 4.2 (continued)

Supported via inline assembly & intrinsic functions

- Intrinsic header file for Nehalem: nmmintrin.h
- automatic generation with /QxSSE4.2 is limited in 11.0

Manual cpu dispatch name: core_i7_sse4_2

e.g.

```
__declspec(cpu_specific(core_i7_sse4_2))

__declspec(cpu_dispatch(core_2_duo_sse4_1, core_i7_sse4_2))
```

Intel® Core™2 Intel® Core™ i7

PCMPGTQ autogeneration example

```
long long dst[NUM], src1[NUM], src2[NUM], src3[NUM], src4[NUM];
 for (i = 0; i < NUM; i++) {
       if (src1[i] <= src2[i]) {
          dst[i] = src3[i];
       } else {
          dst[i] = src4[i];
       }
  }
```

Speedups:
- -Example below: 2.1x
- -MIN/MAX idioms: 2.3x
- -ABS idiom: 2.7x

Vectorization is impossible (without SSE4.2)

```
   xor     eax, eax
$B2$2:
       mov      ecx, DWORD PTR [_src1+eax*8]
       mov      edx, DWORD PTR [_src1+4+eax*8]
       sub      ecx, DWORD PTR [_src2+eax*8]
       sbb      edx, DWORD PTR [_src2+4+eax*8]
       jl       $B2$3
$B2$9:
       or       ecx, edx
       jne      $B2$4
$B2$3:
       mov      edx, DWORD PTR [_src3+eax*8]
       mov      ecx, DWORD PTR [_src3+4+eax*8]
       jmp      $B2$5
$B2$4:
       mov      edx, DWORD PTR [_src4+eax*8]
       mov      ecx, DWORD PTR [_src4+4+eax*8]
$B2$5:
       mov      DWORD PTR [_dst+eax*8], edx
       mov      DWORD PTR [_dst+4+eax*8], ecx
       add      eax, 1
       cmp      eax, 16384
       jl       $B2$2
```

Vectorization is possible with /QxSSE4.2 /Qunroll0

```
   xor      eax, eax
$B2$2:
       movdqa    xmm0, XMMWORD PTR [_src1+eax*8]
       pcmpgtq   xmm0, XMMWORD PTR [_src2+eax*8]
       movdqa    xmm1, XMMWORD PTR [_src3+eax*8]
       pblendvb  xmm1, XMMWORD PTR [_src4+eax*8], xmm0
       movdqa    XMMWORD PTR [_dst+eax*8], xmm1
       add       eax, 2
       cmp       eax, 16384
       jb        $B2$2
```

Autogeneration of STTNI for strlen

Partially inlined implementation

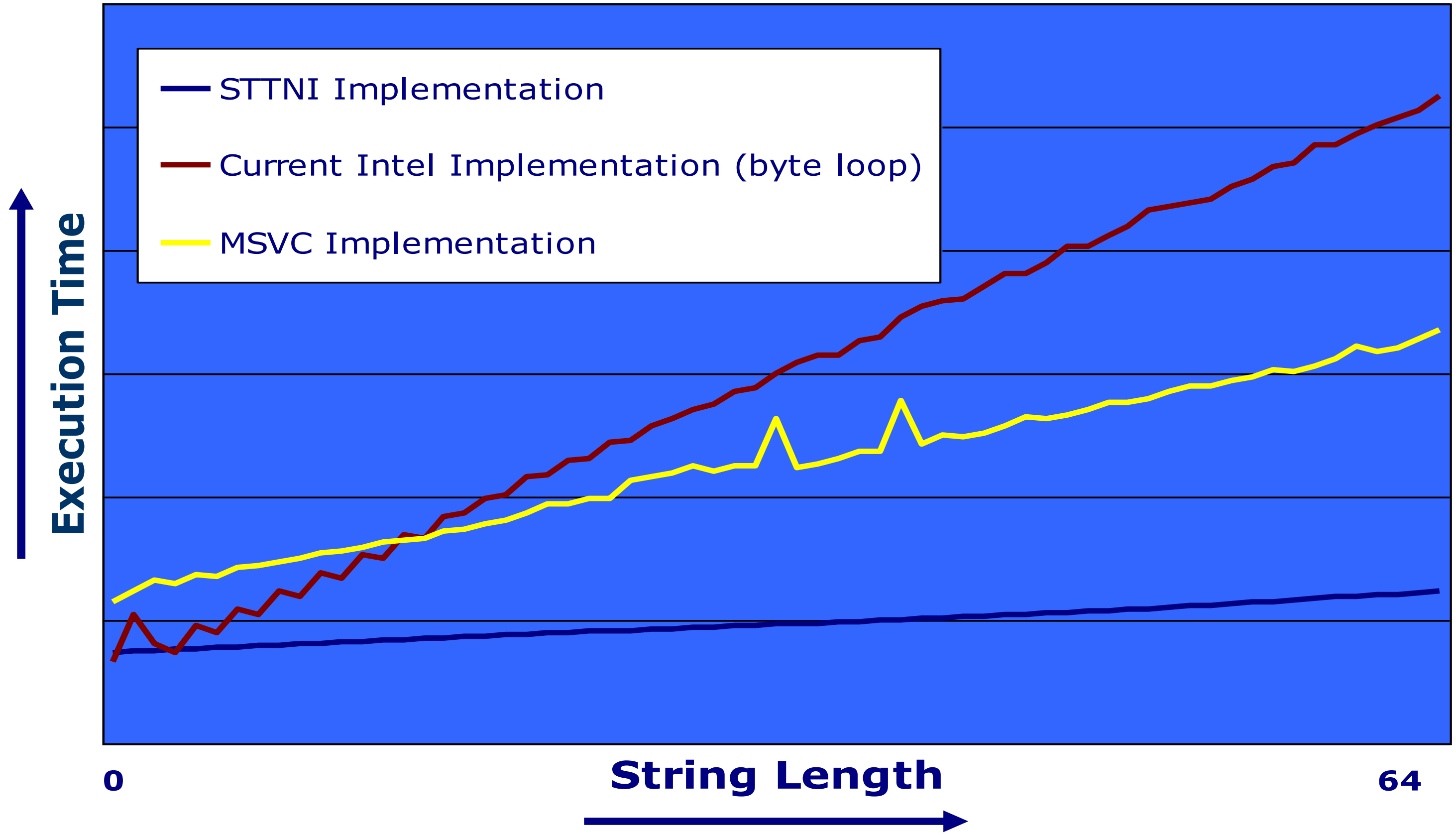
- Avoids call overhead for short strings (common case)
- Avoids the excessive code bloat from fully inlining

```
    mov       ecx, edx
    and       edx, 0xFFFFFFF0
    pxor      xmm0, xmm0
    pcmpeqb   xmm0, XMMWORD PTR [edx]
    pmovmskb  eax, xmm0
    and       ecx, 0xF
    shr       eax, cl
    bsf       eax, eax
    jne       ..L1

    mov       eax, edx
    add       edx, ecx
    call      __intel_sse4_strlen
..L1:
```

```
___intel_sse4_strlen:
    add       eax, 16
    movdqa    xmm0, XMMWORD PTR [eax]
    pcmpistri xmm0, xmm0, 58
    jae       ___intel_sse4_strlen

    sub       ecx, edx
    add       eax, ecx
    ret
```

Autogeneration of STTNI for strlen (in 11.1)

- Comparable performance on short strings
- Over 5x improvement for long strings
- Working on strcpy, strncmp, strcmp implementations



Unaligned Load / Store Improvements

Micro-architectural Feature

- Cache line splits are MUCH less expensive in Nehalem
- Unaligned 16-byte loads/stores are as fast as aligned 16-byte loads/stores when there is no cache line split

Consequence in 11.0 Compiler (with /QxSSE4.2 only) :

- Favor 16-byte unaligned loads (e.g. movups) over multi-instruction sequences designed to avoid potential cache line splits
  - May replace up to 7 instructions
  - Reduces register pressure
  - Don’t do if cache line split is certain
  - 2-3% overall performance benefit on SPEC fp (application-dependent)

CPU2000/CPU2006 Results on Nehalem

CPU2000 Measurements

- No performance regressions
- 168.wupwise +8%
- 172.mgrid +21%
- 178.galgel +3%
- 301.apsi +5%
- **Overall fp Geomean +2.78%**

CPU2006 Measurements

- No performance regressions
- 436.cactusADM +11%
- 437.leslie3d +9%
- 454.calculix +8%
- 459.GemsFDTD +12%
- **Overall fp Geomean +2.6%**

Unaligned Load / Store Improvements

Further compiler opportunities

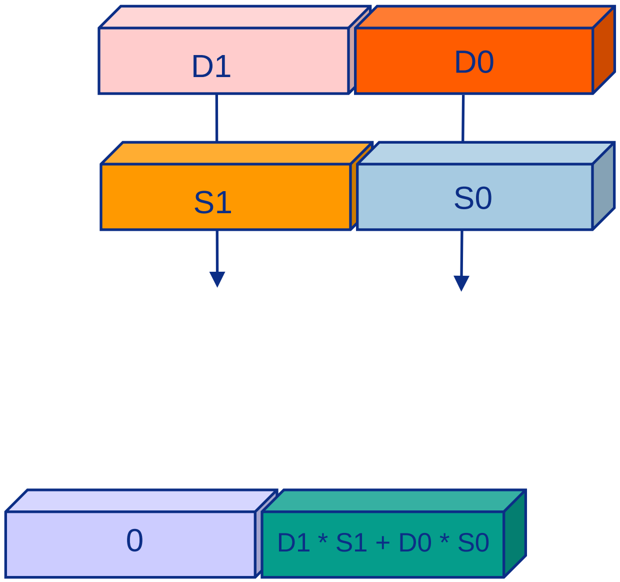
- Vectorize more loops where alignment is not known
- Avoid loop versioning for different relative alignments

Example in 11.0:

- Facilitate use of dppd/dpps (SSE 4.1) when alignment not known
  - Generated for the Fortran DOT_PRODUCT intrinsic when vector length is 4

- However, there are still benefits to aligning data in your code where it is straightforward to do so
  - Avoid cache line splits
  - CISC-ize SSE instructions with memory accesses (i.e., combine load with SSE arithmetic operation in one instruction)
  - 16 byte alignment may become important again for AVX

DPPD / DPPS Tuning



t = a[n] * b[n] + a[n+1] * b[n+1];

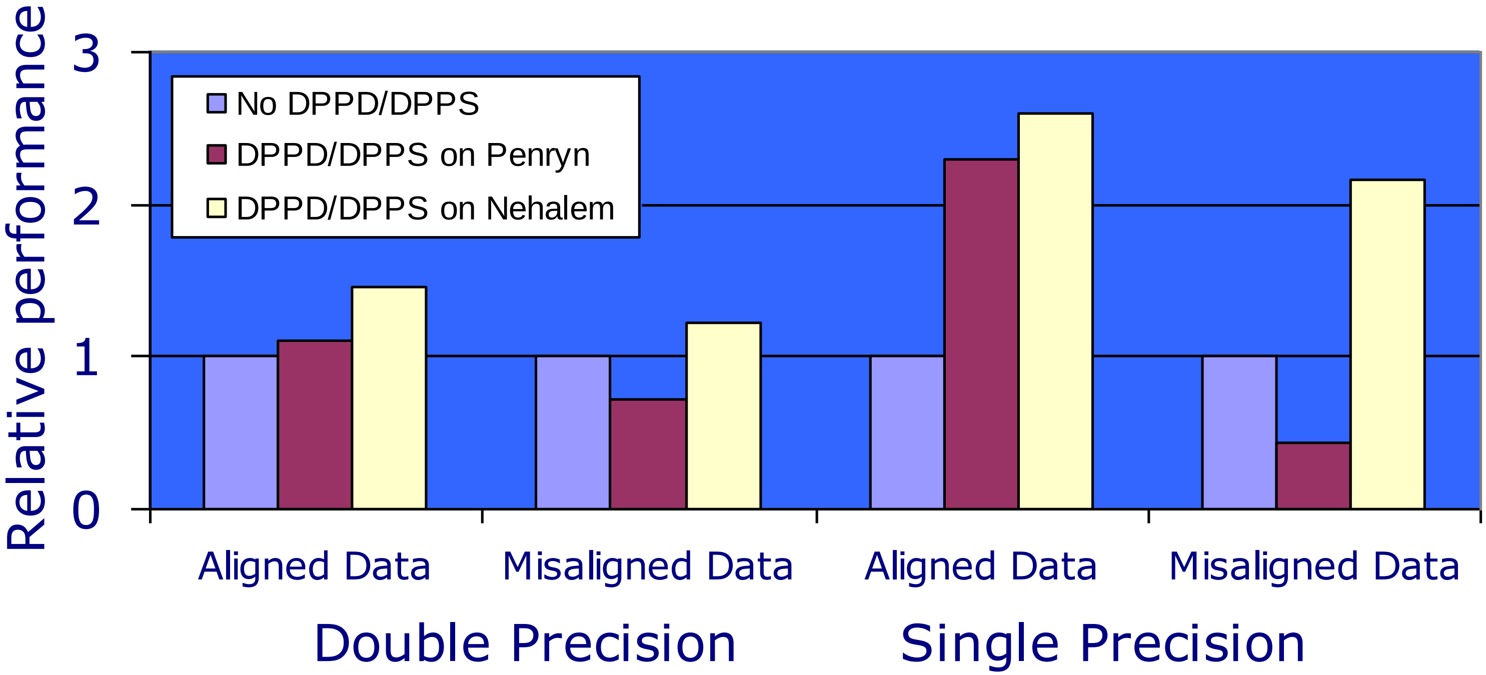
Current heuristics generate split sequence when we cannot prove alignment:

```
movsd      xmm1, QWORD PTR [_a+eax*8]
movhpd     xmm1, QWORD PTR [_a+8+eax*8]
movsd      xmm0, QWORD PTR [_b+eax*8]
movhpd     xmm0, QWORD PTR [_b+8+eax*8]
dppd       xmm1, xmm0, 0x31
```

For Nehalem, we should use

```
movupd     xmm1, XMMWORD PTR [_a+eax*8]
movupd     xmm0, XMMWORD PTR [_b+eax*8]
dppd       xmm1, xmm0, 0x31
```

DPPD / DPPS Tuning



Take advantage of fast unaligned loads

Memory Architectural Changes

Microarchitectural Feature

- Improved memory bandwidth (doesn't need recompile!)
- Integrated memory controller
- Added cache level compared to Intel® Core™ 2
  - 256KB L2 per core, shared L3 ≤8 MB (quad core)
  - "cachesize" intrinsic updated

Compiler Opportunities (potential)

- More aggressive software prefetch (must be done judiciously)
- Library tuning for memset/memcpy
- Blocking, unrolling, etc, for larger cache
- More aggressive auto-parallelization

Some Apps may no longer be memory bound

Memory Architectural Changes

Microarchitectural Feature

- Memory local to each socket (NUMA)
- Simultaneous MultiThreading (SMT)

Compiler Opportunities

- Extended interface for OpenMP thread affinity (done in 11.0)
  - KMP_AFFINITY=compact,1 gives consecutive threads on different physical cores on the same socket, if SMT is **enabled**
  - KMP_AFFINITY=compact gives consecutive threads on different physical cores on the same socket, if SMT is **disabled** (same as compact,0)
  - KMP_AFFINITY=scatter gives consecutive threads on alternating sockets
  - May need KMP_AFFINITY=disable if 3rd party affinity tools used
- Control how memory is allocated between sockets for OpenMP apps (Like memory_touch directive for Intel® Itanium™ processors)

Macrofusion

Microarchitectural Feature

- Processor combines **adjacent** cmp/test + jcc into single uop
  - Increases effective FE & ROB bandwidth
  - More cases supported in NHM over Merom/Penryn
    - Signed jcc conditions
    - Intel 64

Compiler Opportunities

- Already schedules fusible cmp/test + jcc to be adjacent.
- Extend to handle new cases for Intel® Core™ i7 and Xeon™ 5500 processors

Front End

Microarchitectural Issues

- Improved L2->L1 instruction fetch rate
- Increased size of Loop Stream Detector
  - Larger loops are able to fit in the LSD, bypassing the front end and any instruction decoding bottlenecks, & using less power

Compiler Opportunities

- More use of optimizations that result in larger code
  - loop unrolling
  - inlining
- Avoid aligning loops that are likely to be detected by LSD
- More use of instructions that previously would have risked decoding bottlenecks, e.g.
  - LCP instructions like "addw mem16, imm16"
  - POPCNT, et al ?

Backup