Intel® Composer XE

Introduction to Vectorization

Agenda

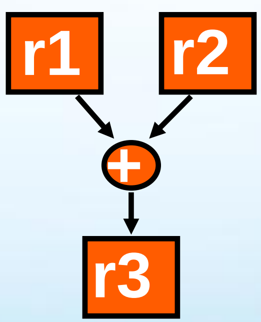
- **Introduction**
- Vector Code Generation
- Compiler Switches for Automatic Vectorization
- Validating Success of Automatic Vectorization
- When Vectorization fails
  - Data Dependence
  - Alignment
  - Others like Non-Unit Stride Access, Function Calls, …
- Vectorization of special program constructs
  - Idiom Recognition
  - Complex data type
- HLO Loop Transformations
- Summary, References

Optimization Notice

Introduction
Vector Processing

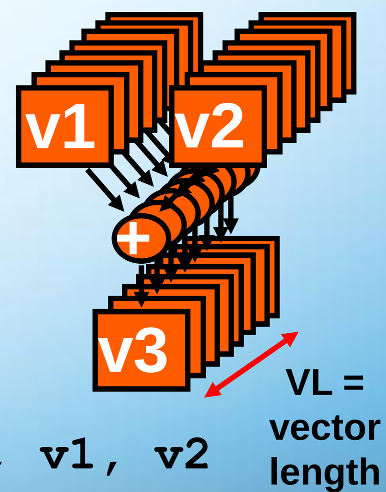
- A specific case of data level parallelism (DLP)
- Same operation simultaneously executed on N >1 elements of a vector – a one-dimensional array of scalar data objects like integers, floats, etc
  - extends scalar processing to parallel execution
- We will call the number of elements in the vector VL (vector length)

Scalar Processing




```
add.d r3, r1, r2
```

Vector Processing



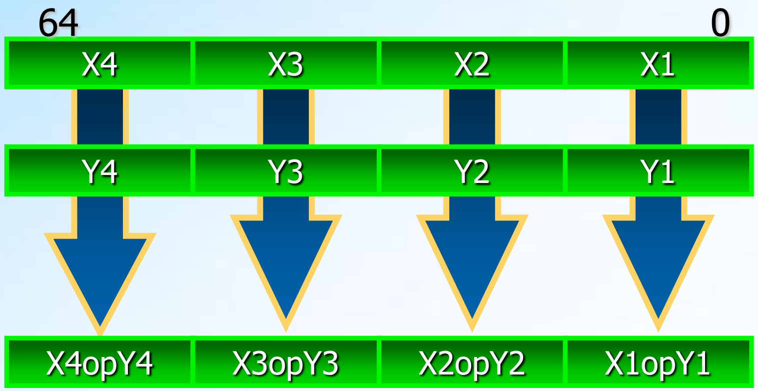

```
addvec.d v3, v1, v2
```

SIMD Execution

- For the architecture we look at vector processing is mainly SIMD (Single Instruction Multiple Data) execution:
  - the vector operation is one single machine instruction
  - the vectors have a fixed short length of VL=2,4,8,16
  - The execution on all elements of the vector is synchronously
    - All results are available at the same time

- SIMD enhancements in processors hardware relevant for our target platforms
  - 64 bit Multi-Media Extension – MMX™
  - 128 bit Intel® Streaming SIMD Extension – Intel® SSE
  - 256 bit Intel® Advanced Vector Extensions – Intel® AVX
  - 512 bit vector instruction set extension of Intel® Many Integrated Core Architecture – Intel® MIC

SIMD Types in Processors from Intel [1]



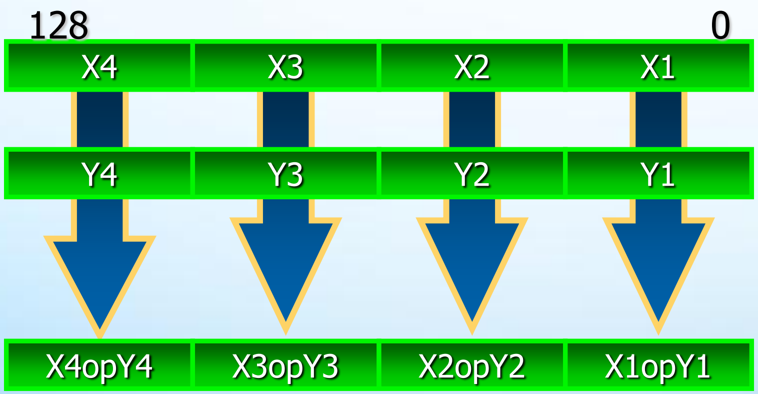
MMX™

Vector size: 64bit

Data types: 8, 16 and 32 bit integers

VL: 2,4,8

For sample on the left: Xi, Yi 16 bit integers



Intel® SSE

Vector size: 128bit

Data types:

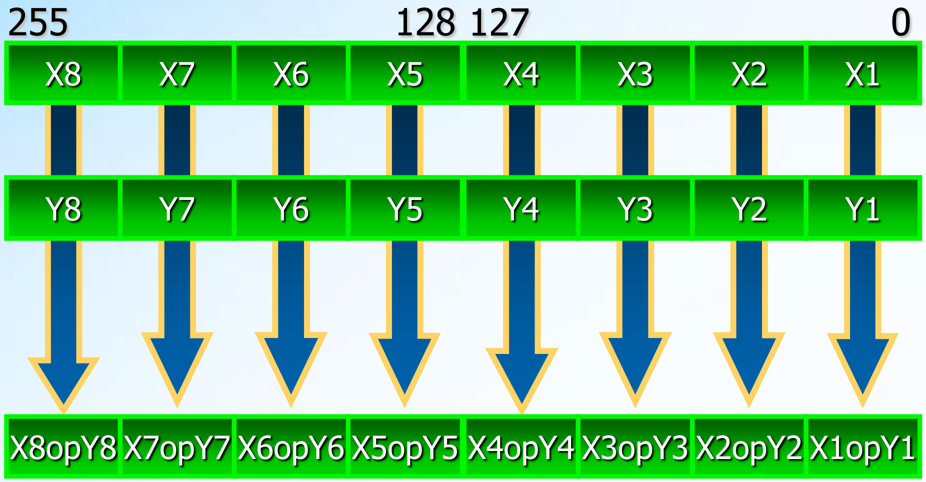
- 8,16,32,64 bit integers
- 32 and 64bit floats

VL: 2,4,8,16

Sample: Xi, Yi bit 32 int / float

Optimization Notice

SIMD Types in Processors from Intel [2]



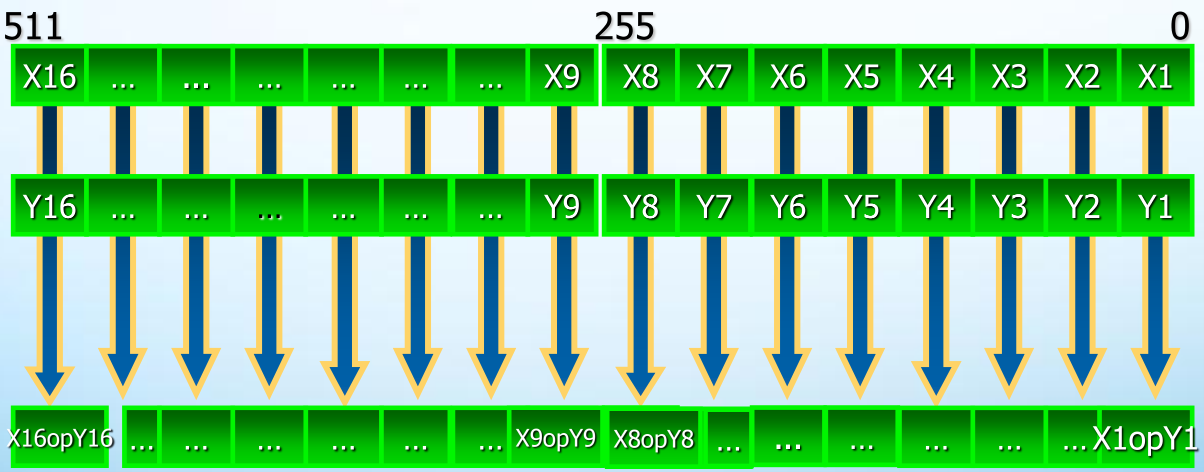
Intel® AVX

Vector size: 256bit

Data types: 32 and 64 bit floats

VL: 4, 8, 16

Sample: Xi, Yi 32 bit int or float



Intel® MIC

Vector size: 512bit

Data types:

32 and 64 bit integers

32 and 64bit floats

(some support for

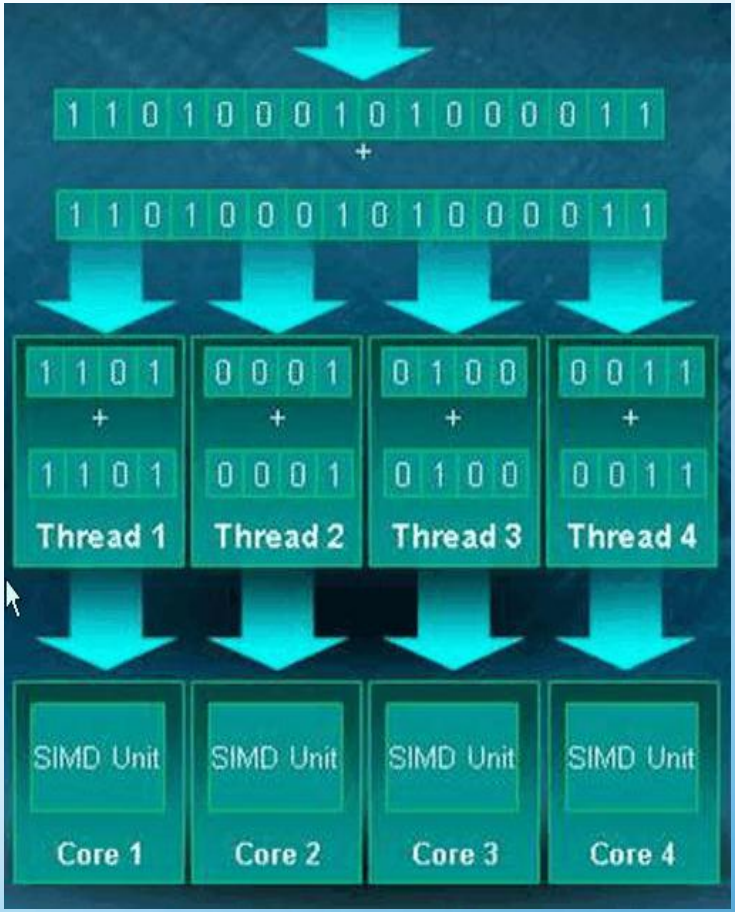
16 bits floats)

VL: 8,16

Sample: 32 bit float

Extending SIMD to Multiple Cores

```
integer a[16],b[16],c[16];
c[:] = a[:] + b[:];
```



- The availability of more and more cores in modern processors offers the opportunity to use these cores to implement vector processing in a new way:
  - Multiple cores simultaneously operate on the data elements of “long” vectors
    - Typically – but not necessarily- all cores execute the same SIMD instruction
  - Semantically this execution model is implemented as a deterministic, race-free execution model – presented to the developer not differently than traditional SIMD processing

Optimization Notice

We will Focus on SSE from now on

- MMX™ has very limited relevance today due to SSE being faster, more flexible, using twice the register size and being available in close to all x86 processors today
  - And MMX-vectorization is not supported anymore in current compilers from Intel
- The content of the following training material is valid for Intel® AVX as it is for Intel® SSE. We will mention some difference where appropriate
- The multi-core vector execution model initially will be exploited by new parallel programming language features & models; for now not by explicit or compiler-guided vectorization

While we will focus on Intel® SSE from now on, the concepts and ideas can be mapped to all the other vector processing models mentioned

Agenda

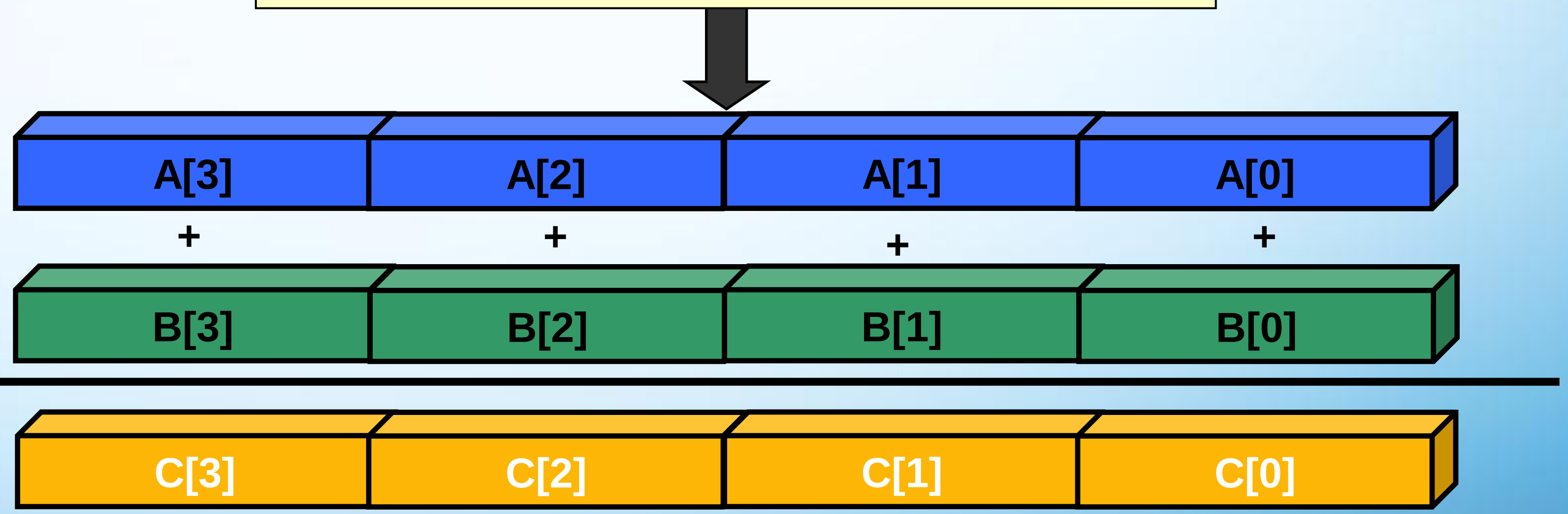
- Introduction
- **Vector Code Generation**
- Compiler Switches for Automatic Vectorization
- Validating Success of Automatic Vectorization
- When Vectorization fails
  - Data Dependence
  - Alignment
  - Others like Non-Unit Stride Access, Function Calls, …
- Vectorization of special program constructs
  - Idiom Recognition
  - Complex data type
- HLO Loop Transformations
- Summary, References

Vectorization

Transforming sequential code to exploit the vector (SIMD, SSE) processing capabilities

- Manually by explicit source code modification
- Automatically by tools like a compiler

```
for (i=0;i<MAX;i++)
    c[i]=a[i]+b[i];
```



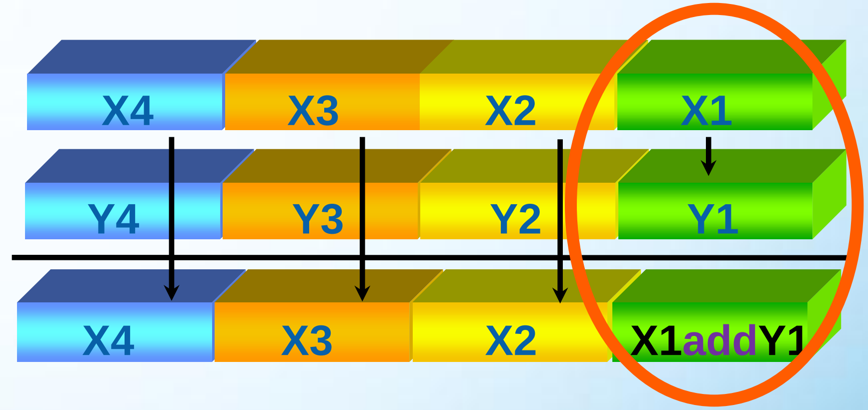
Scalar and Packed SSE Instructions

The “vector” form of SSE instructions operating on multiple data elements simultaneously are called <u>packed</u> – thus vectorized SSE code means use of packed instructions

- Most of these instructions have a <u>scalar</u> version too operating only one element only

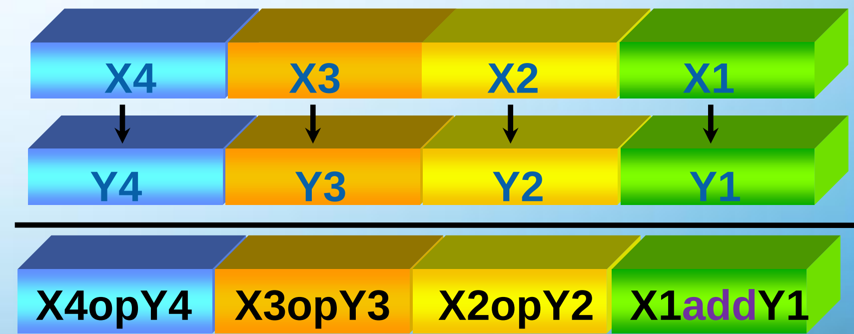
addss Scalar Single-FP Add

single precision FP data
scalar execution mode



addps Packed Single-FP Add

single precision FP data
packed execution mode



Two Key Decisions to be Made :

1. How do we introduce the vector code ?
   - Our key focus will be **automatic vectorization** by the Intel® Compilers but there are other ways too
2. How do we deal with the multiple SIMD instruction set extensions like SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX …?
   - Each instruction set extension includes all others released before. Thus we should use the latest one supported
     - In case we know the target platform to have one specific processor model, it is a simple decision
     - Otherwise, **run-time processor dispatching** should be an option to select the appropriate path for the given architecture

Many Ways for SSE Vectorization

Compiler: Fully automatic vectorization

Compiler: Auto vectorization hints (#pragma ivdep, …)

SIMD intrinsic class (F32vec4 add)

Vector intrinsic (mm_add_ps())

Assembler code (addps)

Ease of use

Programmer control

Refresh: Intel Instruction Set Extensions

1999	2000	2004	2006	2007	2008
SSE	SSE2	SSE3	SSSE3	SSE4.1	SSE4.2
70 instr Single-Precision Vectors Streaming operations	144 instr Double-precision Vectors 8/16/32 64/128-bit vector integer	13 instr Complex Data	32 instr Decode	47 instr Video Graphics building blocks Advanced vector instr	8 instr String/XML processing POP-Count CRC

Continued by

- **Intel® AES New Instructions - Intel® AES-NI (2009)**
- **Intel® Advanced Vector Extensions – Intel® AVX (2010/11)**

Optimization Notice

Selecting Right Extensions makes a Difference !

```
static double A[1000], B[1000],
              C[1000];
void add() {
  int i;
  for (i=0; i<1000; i++)
    if (A[i]>0)
      A[i] += B[i];
    else
      A[i] += C[i];
}
```

SSE2

```
.B1.2::
  movaps     xmm2, A[rdx*8]
  xorps      xmm0, xmm0
  cmpltpd    xmm0, xmm2
  movaps     xmm1, B[rdx*8]
  andps      xmm1, xmm0
  andnps     xmm0, C[rdx*8]
  orps       xmm1, xmm0
  addpd      xmm2, xmm1
  movaps     A[rdx*8], xmm2
  add        rdx, 2
  cmp        rdx, 1000
  jl         .B1.2
```

AVX

```
.B1.2::
  vmovaps   ymm3, A[rdx*8]
  vmovaps   ymm1, C[rdx*8]
  vcmpgtpd  ymm2, ymm3, ymm0
  vblendvpd ymm4, ymm1,B[rdx*8], ymm2
  vaddpd    ymm5, ymm3, ymm4
  vmovaps   A[rdx*8], ymm5
  add       rdx, 4
  cmp       rdx, 1000
  jl        .B1.2
```

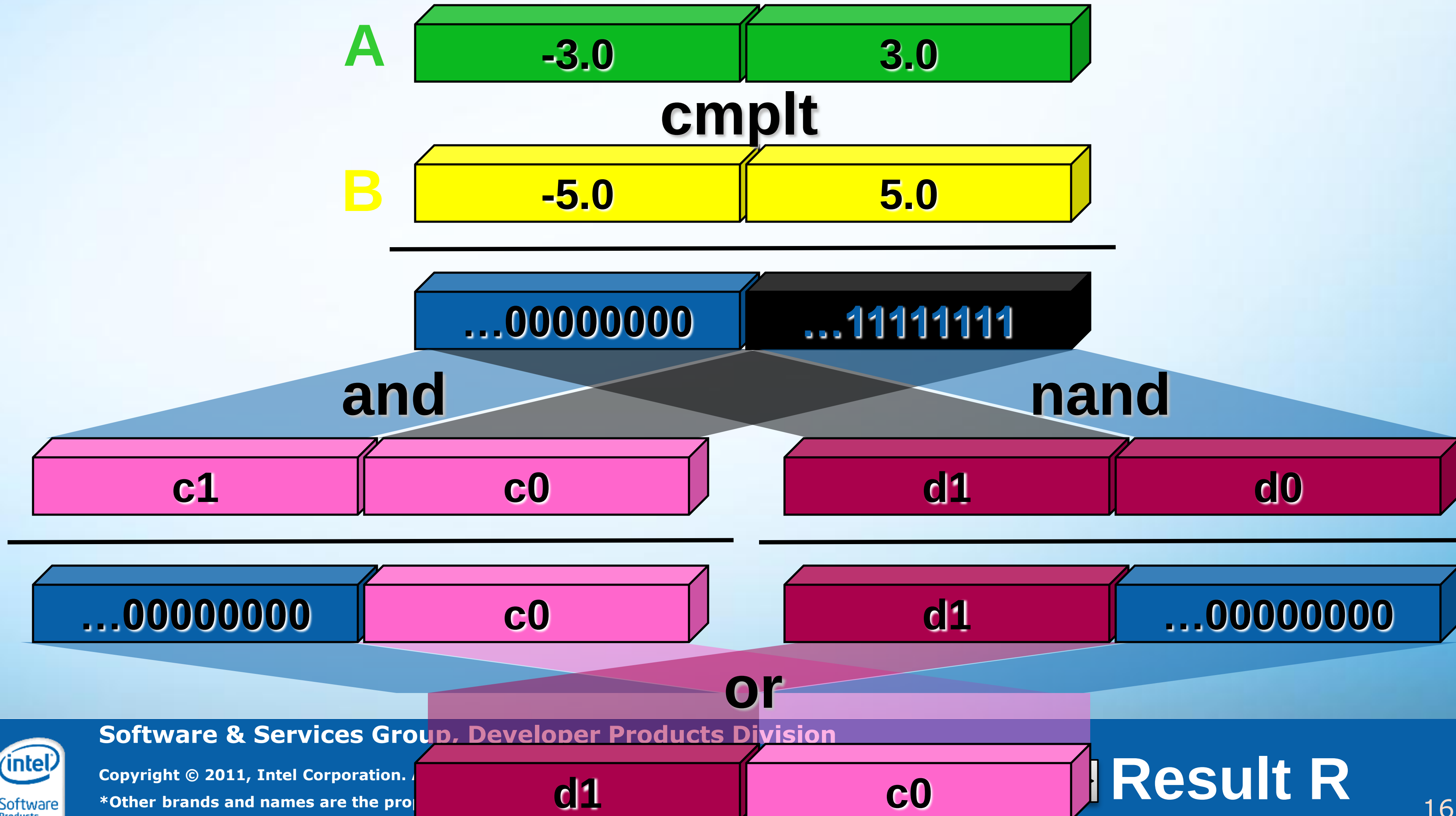
SSE4.1

```
.B1.2::
  movaps     xmm2, A[rdx*8]
  xorps      xmm0, xmm0
  cmpltpd    xmm0, xmm2
  movaps     xmm1, C[rdx*8]
  blendvpd   xmm1, B[rdx*8], xmm0
  addpd      xmm2, xmm1
  movaps     A[rdx*8], xmm2
  add        rdx, 2
  cmp        rdx, 1000
  jl         .B1.2
```

Sample for using SSE Intrinsics

Conditions without Jumps

```
for (i=0,…) R[i] = (A[i]<B[i])? C[i]:D[i];
```



Implementation Using Intrinsics

```
// R[i] = (A[i] < B[i])? C[i] : D[i]

__m128d mask = _mm_cmplt_pd(a, b);
r = _mm_or_pd(
    _mm_and_pd (mask, c),
   _mm_nand_pd(mask, d)
);
```

- Intrinsics or SIMD Vector Classes should be preferred to explicit assembler coding
  - Similar performance, very close to best manually written assembler code
  - Hides many details like register allocation and scheduling
  - Intrinsics more portable and supported by all popular compilers !

Manual Processor Dispatch

- Intel® C++ Compiler provides API to implement one function in specific, explicit versions for multiple Intel® processors architectures
- The processor architectures are identified by a cpuid-keyword like *core_i7_sse4_2* for the Intel® Core™ i7 processor architecture
- Two extensions to function declarations:
  - To define the routine being ‘dispatched’ and the processor architecture list:

    `__declspec( cpu_dispatch(`*`cpuid-list`*`) ) func(..)`
  - To define the individual implementations:

    `__declspec( cpu_specific(`*`cpuid`*`) ) func(..)`
- For more details and example, see article on software.intel.com

Optimization Notice

Agenda

- Introduction
- Vector Code Generation
- **Compiler Switches for Automatic Vectorization**
- Validating Success of Automatic Vectorization
- When Vectorization fails
  - Data Dependence
  - Alignment
  - Others like Non-Unit Stride Access, Function Calls, …
- Vectorization of special program constructs
  - Idiom Recognition
  - Complex data type
- HLO Loop Transformations
- Summary, References

Compiler Based Vectorization

Extension Specification

Feature	Extension
Intel® Streaming SIMD Extensions 2 (Intel® SSE2) as available in initial Pentium® 4 or compatible non-Intel processors	SSE2
Intel® Streaming SIMD Extensions 3 (Intel® SSE3) as available in Pentium® 4 or compatible non-Intel processors	SSE3
Intel® Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) as available in Intel® Core™2 Duo processors	SSSE3
Intel® SSE4.1 as first introduced in Intel® 45nm Hi-K next generation Intel Core™ micro-architecture	SSE4.1
Intel® SSE4.2 Accelerated String and Text Processing instructions supported first by by Intel® Core™ i7 processors	SSE4.2
Extensions offered by Intel® ATOM™ processor : Intel® SSSE3 (!!) and MOVBE instruction	SSE3_ATOM
Intel® Advanced Vector Extensions (Intel® AVX) as available in 2nd generation Intel Core processor family	AVX

Basic Vectorization – Switches [1]

{L&M} -x<extension> {W}: /Qx<extension>

- Targeting Intel® processors - specific optimizations for Intel® processors
- Compiler will try to make use of all instruction set extensions up to and including <extension>; for Intel® processors only !
- Processor-check added to main-program
- Application will not start (will display message), in case feature is not available

{L&M}: -m<extension> {W}: /arch:<extension>

- No Intel processor check
- Does not perform Intel-specific optimizations
- Application is optimized for and will run on both Intel and non-Intel processors
- Missing check can cause application to fail in case extension not available

{L&M}: -ax<extension> {W}: /Qax<extension>

- Dual-code paths – a 'generic' and 'optimized' path
- 'processor-specific' path for Intel® processors defined by <extension>
- 'default' code path defaults to –msse2 (Windows: /arch:SSE2)
  - The 'default' code path can be modified by –m or –x (/Qx or /arch) switches

Basic Vectorization – Switches [2]

The default now is –msse2 (Windows: /arch:SSE2)

- Activated implicitly for –O2 or higher
- Implies the need for a target processor with Intel® SSE2
- Use –mia32 (Windows /arch:IA32) in case target processor misses SSE2 (Intel® Pentium™ 3 processor for example)

Special switch -xHost (Windows: /QxHost)

- Compiler checks host processor and makes use of 'latest' instruction set extension available
- Avoid for builds being executed on multiple, unknown platforms

Some support for combination of -x<ext1> and -ax<ext2> switches (Windows: /Qx<ext1> and /Qax<ext2>)

- Can result in more than 2 code paths
- Use ext1 = ia32 in case 'generic' code path should support too very early processors not supporting SSE2 (e.g. Intel® Pentium™ 3)

Vectorization – More Switches and Directives

Disable vectorization

- Globally via switch: {L&M}: `-no-vec` {W}: `/Qvec-`
- For a single loop: directive `novector`
  - Disabling vectorization here means not using packed SSE/AVX instructions. The compiler still might make use of the corresponding instruction set extensions

Enforcing vectorization for a loop - overwriting the compiler heuristics : `#pragma vector always`

- will enforce vectorization even if the compiler thinks it is not profitable to do so (e.g due to non-unit strides or alignment issues)
- Will not enforce vectorization if the compiler fails to recognize this as a semantically correct transformation
- Using directive `#pragma vector always assert` will print error message in case the loop cannot be vectorized and will abort compilation

Vectorization Switches – Some Notes

- Former vectorization switches like –xW, /QxT, /QaxP etc are considered ‘deprecated’ and will not be supported anymore in the future
  - See appendix or compiler documentation for mapping between old and new names
- It is not possible anymore to generate vector code exclusively for the initial SSE (32 bit FP) instruction set (introduced by Intel® Pentium™ 3 processor)
- The instruction set extension name for Intel® Atom™ processors (SSE3_ATOM) is misleading: Since the architecture supports up to Intel® SSSE3, e.g. switch -xSSE3_ATOM will make use of SSSE3 too
  - In case the code should be optimized for Intel® Atom processors but should run too on all processors supporting up to SSSE3, add option –minstruction=nomovbe (Windows: /Qinstruction:nomovbe) to avoid the use of the Atom-specific instruction MOVBE

Student Exercise # 1

Which Loops will Vectorize ?

```
#01: for (j=1; j<MAX; j++)  a[j]=a[j-n]+b[j];

#02: for (int i=0; i<SIZE; i+=2)  b[i] += a[i] * x[i];

#03: for (int j=0; j<SIZE; j++)
       for (int i=0; i<SIZE; i++)
            b[i] += a[i][j] * x[j];

#04: for (int i=0; i<SIZE; i++)
       b[i] += a[i] * x[index[i]];

#05: for (j=1; j<MAX; j++)  sum = sum + a[j]*b[j]

#06: for (int i=0; i<length; i++)
       if ( s >= 0 )
          x2[i] = (-b[i]+sqrt(s))/(2.*a[i]);
```

Student Exercise

Sample program showing effects when compiling for difference instruction set extensions

Agenda

- Introduction
- Vector Code Generation
- Compiler Switches for Automatic Vectorization
- **Validating Success of Automatic Vectorization**
- When Vectorization fails
  - Data Dependence
  - Alignment
  - Others like Non-Unit Stride Access, Function Calls, …
- Vectorization of special program constructs
  - Idiom Recognition
  - Complex data type
- HLO Loop Transformations
- Summary, References

Validating Vectorization Success

- Assembler code inspection
  - Assembler listing: {L&M}: –S and {W}: /Fa
  - Most reliable way and gives all details of course
  - Check for scalar or packed instructions
    - Assembler listing contains source line numbers mapping generated code to loops in source code
- Optimization report of "High-Performance-Optimizer" (HPO) phase
  - `{L&M}: -opt-report<N>    -opt-report-phasehpo`
  - `{W}:   /Qopt-report:<N>  /Qopt-report-phase:hpo`
    - N=1,2,3 specifies level of detail, N=2 is default
  - We will come back to the opt-report switch later again
- Vectorization report
- Dynamically counting the number of executed **packed** SSE instructions using tools like Intel® VTune Amplifier™ profiler
  - E.g. using performance monitoring event FP_COMP_OPS_EXE.SSE_FP_PACKED on Intel® Core™ i7 processors

Vectorization Report

- Provides details on vectorization success & failure
  - L&M: -vec-report<n>, n=0,1,2,3,4,5
  - W: /Qvec-report<n>, n=0,1,2,3,4,5

```
35:     subroutine fd( y )
36:     integer :: i
37:     real, dimension(10), intent(inout) :: y
38:     do i=2,10
39:        y(i) = y(i-1) + 1
40:     end do
41:     end subroutine fd
```

```
novec.f90(38): (col. 3) remark: loop was not vectorized: existence of
vector dependence.
novec.f90(39): (col. 5) remark: vector dependence: proven FLOW
dependence between y line 39, and y line 39.
novec.f90(38:3-38:3):VEC:MAIN_:  loop was not vectorized: existence of
vector dependence
```


Optimization Notice

Diagnostic Level of Vectorization Switch
L&M: -vec-report<N> W: /Qvec-report<N>

N	Diagnostic Messages
0	No diagnostic messages; same as not using switch and thus default
1	Report about vectorized loops– default if switch is used but N is missing
2	Report about vectorized loops and non-vectorized loops
3	Same as N=2 but add add information on assumed and proven dependencies
4	Report about non-vectorized loops
5	Same as N=4 but add detail on why vectorization failed

Note:

– In case inter-procedural optimization (-ipo or /Qipo) is activated and compilation and linking are separate compiler invocations, the switch needs to be added to the link step

Optimization Notice

Agenda

- Introduction
- Vector Code Generation
- Compiler Switches for Automatic Vectorization
- Validating Success of Automatic Vectorization
- **When Vectorization fails**
  - **Data Dependence**
  - Alignment
  - Others like Non-Unit Stride Access, Function Calls, …
- Vectorization of special program constructs
  - Idiom Recognition
  - Complex data type
- HLO Loop Transformations
- Summary, References

When Vectorization Fails …

- Most frequent reason: Dependence
  - Simplified: Loop iterations must be independent
- Many other potential reasons
  - Alignment
  - Function calls in loop block
  - Complex control flow / conditional branches
  - Loop not "countable"
    - E.g. upper bound no run time constant
  - Not inner loop
    - Outer loop of nest cannot be vectorized
  - Mixed data types (many cases now handled successfully)
  - Non-unit stride between elements
  - Loop body too complex– register pressure
  - Vectorization seems inefficient
  - Many more … but less likely too occur

Dependence Terminology

Dependence is a key term for vectorization:

- Vectorization is a transformation changing the execution order of statements
- The execution order of statements as defined by the program source code (“textual order”) can be changed as long as the dependencies between all statements are preserved

A dependence either is a data or control dependence

```
S1   A = 3.0
S2   B = 4.0
S3   C = sqrt(A**2, B**2)
```

Data dependence from S_1 to S_3 and from S_2 to S_3

```
S1   if ( T != 0 )
S2     A = A / T
```

Control dependence from S_1 to S_2

Control dependencies in loops frequently can be converted to data dependencies or can be eliminated completely - we will come back to this later.

Data Dependence

Definition of data dependencies

- There is a data dependence from statement S_1 to S_2 statement (written $S_1 \, \delta \, S_2$) **if and only if :**
  - There is a potential execution flow from S_1 to S_2
  - S1 and S2 reference a common memory location and either S_1 or S_2 write to it
- Note: S_1 and S_2 can be the very same statement

Data dependence classification:

$S_1 \, \delta^F \, S_2$: S_1 writes, S_2 reads : Flow Dependence

$S_1 \, \delta^A \, S_2$: S_1 reads, S_2 writes : Anti Dependence

$S_1 \, \delta^O \, S_2$: S_1 writes, S_2 writes: Output Dependence

S_1	**X** = …
S_2	… = **X**

$S_1 \, \delta^F \, S_2$

S_1	… = **X**
S_2	**X** = …

$S_1 \, \delta^A \, S_2$

S_1	**X** = …
S_2	**X** = …

$S_1 \, \delta^O \, S_2$

Optimization Notice

Data Dependence in Loops

Dependencies in loops are most interesting for us since vectorization almost exclusively is applied to loops

– Dependencies in loops become more obvious by virtually unrolling the loop:

$S_1 \, \delta^F S_1$

```
     DO I = 1, N
S1:    A(I+1) = A(I) + B(I)
     ENDDO
```

```
S1  A(2) = A(1) + B(1)
S1  A(3) = A(2) + B(2)
S1  A(4) = A(3) + B(3)
S1  A(5) = A(4) + B(4)
     …
```

In case the dependency requires execution of more than one loop iteration to exist, we call it *loop-carried dependence*. Otherwise *loop-independent dependence*

```
   DO I = 1, 10000
S1      A(I) = B(I) * 17

S2      X(I+1) = X(I) + A(I)
   ENDDO
```

($S_1 \, \delta^F \, S_2$) is a loop-independent dependence

($S_2 \, \delta^F \, S_2$) is loop-carried dependence

Optimization Notice

Student Exercise # 2

Find (if any) all Dependencies in these Samples

Dependencies? Type ?

```
for (i=0;i<MAX-2,i++)
S:   A[i+2]=A[i] + 1;
```

```
for (i=1;i<MAX,i++)
{
S1:  A[i]=A[i-1] * 2;
S2:  B = A[i-1];
}
```

```
for (i=0;i<MAX,i++)
S: A[i+1,j] = A[i,k] + B;
```

```
for (i=0;i<MAX-1,i+=2)
S:  A[i+1]=A[i] + 1;
```


Optimization Notice

Dependence & Vectorization

- Vectorization of a loop is similar to parallelization - executing the loop iterations in parallel (e.g. via OpenMP threads). However it is not identical:
  - Parallelization requires all iterations to be independent: Thus loop-carried dependencies are not permitted; loop-independent dependencies are ok
  - Vectorization is applied individually to each instruction of the loop body. That is we execute the first instruction in parallel for multiple iterations, then the second in parallel for multiple iterations, …
    - For a loop body with a single statement only, this is identical to parallelization
    - In case we have multiple instructions, it can be very different however

```
DO I=1,N
  A(I+1) = B(I) + C
  D(I) = A(I) + E
END DO
```

Loop can not be *parallelized* but can be *vectorized* to :

```
A(2:N+1) = B(1:N)+C
D(1:N) = A(1:N) + E
```


Optimization Notice

Key Theorem for Vectorization

A loop can be vectorized **if and only if** there is no cyclic dependency chain between the statements of the loop body

- For the formal proof, we refer to the literature – see reference [3]
- The theorem takes into account, that certain semantic-preserving reordering transformations can be applied (e.g. loop distribution)
- The theorem assumes an “unlimited” vector length VL. In case VL is fixed to some constant 2,4,8, … like we have for SSE/AVX, loop carried dependencies requiring VL+1 or more iterations to exist, might be ignored.
  - Thus in some cases vectorization **for SSE/AVX** might be valid while the theorem says no !

Sample:

```
DO I=1,N
  A(I) = A(I+3) + C
END DO
```

Although we have a cyclic dependency chain, the loop can be *vectorized* for SSE in case data type is double precision float but not for single precision float

Optimization Notice

Dealing with Dependencies #1
Hints to the Compiler

- Many dependencies assumed by compiler are false dependencies caused by unresolved memory disambiguation
  - The compiler has to be conservative and has to assume the worst case regarding "aliasing"

```
// Sample: Without additional information (like inter-procedural
// knowledge) compiler has to assume 'a' and 'b' to alias
void scale(int *a, int *b)
{
    for (int i=0; i<10000; i++) b[i] = z*a[i];
}
```

- Many directives, switches and attributes to pass "disambiguation hints" to compiler
  - Programming language and operating system specific
  - Use with care: The compiler might generate incorrect code in case the hints are not fulfilled !

Disambiguation Hints
The "restrict" Keyword for Pointers

{L&M}: -restrict {W}: /Qrestrict

{L&M}: -std=c99 {W}: /Qstd=c99

– Assertion to compiler, that only the pointer or a value based on the pointer - such as (pointer+1) - will be used to access the object it points to

– Only available for C, not C++

```
void scale(int *a, int * restrict b)
{
    for (int i=0; i<10000; i++) b[i] = z*a[i];
}

// two-dimension example:
void mult(int a[][NUM],int b[restrict][NUM]);
```

Disambiguation Hints [C/C++]

A few Selected Directives and Switches

IVDEP directive

- "Ignore Vector Dependencies" - compiler will ignore assumed but not proven dependencies for loop following directive
- In case used together with switch –ivdep-parallel (/Qivdep-parallel), only loop-carried dependencies are ignored

Assume no aliasing at all

- {L&M}: -fno-alias {W}: /Oa

Assume ISO C Standard aliasing rules

- {L&M}: -ansi-alias {W}: /Qansi-alias
- A pointer can be de-referenced only to an object of the same type or compatible type

No aliasing for function arguments

- {L&M}: -fargument-noalias {W}: /Qalias-args-
- For each given function, the arguments of this function don't refer to a common memory object

Disambiguation Hints [Fortran]

A few Selected Directives and Switches

IVDEP directive

- "Ignore Vector Dependencies" - compiler will ignore assumed but not proven dependencies for loop following directive
- In case used together with switch –ivdep-parallel (/Qivdep-parallel), only loop-carried dependencies are ignored
  - In Fortran this is identical to "cDEC$ IVDEP:LOOP"

Assume no aliasing at all

- {L&M}: -fnoalias {W}: /Fa

Assume Fortran Standard aliasing rules

- {L&M}: -ansi-alias {W}: /Qansi-alias
- Different from C/C++, this is default
- the semantic is different from C/C++ and not only cover pointers

No aliasing for function arguments

- {L&M}: -fargument-noalias {W}: /Qalias-args-
- For each given function, the arguments don't alias

No aliasing of "Cray-Pointers"

- {L&M}: -safe-cray-pointers {W}: /Qsafe-cray-pointers

Dealing with Dependencies #2

- Dynamic data dependency analysis
  - The compiler can (!) use run-time checks to test for aliasing
    - E.g. Array A[La:Ua], B[Lb:Ub] overlap ⇔ La<Ub && Lb<Ua
  - The outcome of test is used to execute a vectorized or scalar version of the loop ("Loop Versioning")
  - The heuristic of compiler implements a balance between overhead of testing and performance gains
    - E.g. for an assignment

      $$A[..] = B_1[\ldots] + B_2[\ldots] + \ldots + B_N[\ldots]$$

      the versioning might be done for N=2 but not for N=5
    - Use switch -opt-multi-version-aggressive (/Qopt-multi-version-aggressive for Windows) to change heuristic
- Inter-procedural Dependency Analysis
  - Can improve dependence analysis accuracy considerably
  - Activated by "inter-procedural optimization": -ipo (/Qipo for Windows)
    - For optimization level 2, 3, file-local IPO is on by default
  - Definitions & allocation of function arguments might become visible
  - In case loop body has function call, references in the called function to global variables and actual arguments can be analyzed

Agenda

- Introduction
- Vector Code Generation
- Compiler Switches for Automatic Vectorization
- Validating Success of Automatic Vectorization
- **When Vectorization fails**
  - Data Dependence
  - **Alignment**
  - Others like Non-Unit Stride Access, Function Calls, …
- Vectorization of special program constructs
  - Idiom Recognition
  - Complex data type
- HLO Loop Transformations
- Summary, References

Alignment

- In general, the memory accesses in packed SSE instructions require the data to be aligned to 16 byte boundaries
- For packed AVX instructions, it has to be 32 byte alignement
- Unaligned data can be moved to XMM(YMM) registers using "unaligned load/store" instructions
  - However these instruction are very slow except for SSE memory operations on Intel® Core™ i7 processors or processors based on future Sandy Bridge architecture
- The compiler splits expensive unaligned memory operations into 2 partial loads/stores (e.g. two 64byte loads for one 128byte unaligned load) since this is faster – but still much more expensive than the aligned moves
- The compiler can use 'versioning' in case alignment is unclear
  - A run time check tests for alignment controls execution of a fast version of the loop assuming required alignment or a slower one assuming unaligned data

Alignment Hints to Compiler [C/C++]

- Aligned heap memory allocation by intrinsic / library call

```
void* _mm_malloc (int size, int base)
```

  Linux & Mac OS X only:

```
int posix_memaligned(void **p,size_t base,size_t size)
```

- Directive to assert to compiler, that aligned memory operations can be used for all data accesses in loop following directive

```
#pragma vector aligned | unaligned
```

  - Use with care: The assertion must be satisfied not only by start addresses of all arrays used in loop but for all (!!) data accesses
- Align attribute for variable declarations

```
{W,L,M}: __declspec(align(base)) <array_decl>
{L&M}: <array_decl> __attribute__((aligned(base)))
```

  - The declspec-notation for Linux/Mac OS X is an Intel-specific extension – not working for the GCC compiler. For pure Linux/Mac OS X development, the equivalent attribute-syntax should be preferred
- Assertion to compiler that in the loop following the start address of an array can be assumed to be aligned
  - A language extension (not a directive) for C/C++

```
__assume_aligned(<variable>,base)
```

Alignment Hints to Compiler [Fortran]

- Directive to assert to compiler, that aligned memory operations can be used for all data accesses in loop following directive

```
cDEC$ [ VECTOR ALIGNED | UNALIGNED ]
```

  - Use with care: The assertion must be satisfied not only by start addresses of all arrays used in loop but for all (!!) data accesses
- Assertion to compiler that in the loop following the start address of an array can be assumed to be aligned
  - A directive for Fortran

```
cDEC$ ASSUME_ALIGNED variable:base
```

- Align array definition

```
cDEC$ ATTRIBUTES ALIGN: base :: variable
```

Alignment can be tricky …

```
void matvec(double a[][COLWIDTH], double b[], double x[])
{
  int i, j;
  for (i = 0; i < size1; i++) {
    b[i] = 0;
    #pragma vector aligned
    for (j = 0;j < size2; j=j++)
      b[i] += a[i][j] * x[j];
  }
}
```

- Let us assume, a, b, c would be declared 16-byte aligned in calling routine
- Would this be correct when compiled for SSE2 ?
- Depends on COLWIDTH
  - In case it is even : All ok !
  - In case it is odd: The generated, vectorized code would fail by alignment error !
- Using `__assume_aligned(a,n)` is legal since this refers to the start address only. It wouldn't change much for the vectorization however

Alignment Improvements for Intel® Core™ i7 and Future Processors from Intel

- Unaligned data moves of 128 bytes SSE data are as fast as the aligned versions
- One (unaligned) instruction on the new processors can replace sequences of up to 7 on previous architectures
  - Fewer instructions, better use of instruction-cache, less power consumption and faster code !!!
- To get this benefit, the corresponding extension switch has to be used (e.g. –xSSE4.2 or –xAVX)
- Please note however:
  - Aligned move on un-aligned data (e.g. SSE intrinsics) still fails !
  - Future Sandy Bridge architecture will offer this advantage too for 128 byte data moves, for 256 byte data operation we continue to face alignment challenges !

Alignment Improvements - Example

```
void p(int n, double* s1, double* s2, double* s3, double* dst)
{
  for (int i=0;i<n;i++) dst[i] = s1[i] * s2[i+1] + s3[i+1];
}
```

n=100000, 10000 calls

Intel® XEON X5560 EP System, 2.8 GHz

Linux Redhat 5.3

Intel Compiler 12.0-048 (12.0 Beta Update 2)

icc –O3 –xSSE2   –fno-alias : 1.04 seconds, 4.9*10**9 instructions in p()

icc –O3 –xSSE4.2 –fno-alias : 0.66 seconds, 3.4*10**9 instructions in p()

Very artificial case – does not reflect average gain
but shows potential alignment benefit

Optimization Notice

Agenda

- Introduction
- Vector Code Generation
- Compiler Switches for Automatic Vectorization
- Validating Success of Automatic Vectorization
- **When Vectorization fails**
  - Data Dependence
  - Alignment
  - **Others like Non-Unit Stride Access, Function Calls, …**
- Vectorization of special program constructs
  - Idiom Recognition
  - Complex data type
- HLO Loop Transformations
- Summary, References

Unsupported Loop Structure

- Unsupported loop structure frequently means, the compiler can't construct a runtime expression for the trip-count
  - E.g. a while-loop where the number of iterations cannot be determined at (run-time) start of loop
  - Upper/lower bound of a for-loop cannot be a determined to be loop-invariant
- Frequently this can fixed by minor modifications:

```
struct _x { int d; int bound;};

doit1(int *a, struct _x *x)
{
  for (int i=0; I < x->bound; i++)
    a[i] = 0;
}
```

```
struct _x { int d; int bound;};

doit1(int *a, struct _x *x)
{
  int local_ub = x->bound;
  for (int i=0; I < local_ub; i++)
    a[i] = 0;
}
```

Non-Unit Stride Access

Non-unit stride access: Nonconsecutive memory locations are being accessed in the loop

- Vectorization might still be possible (e.g. in case access is regular/linear), the data arrangement operations might be too expensive
  - Vector report: **"Loop was not vectorized: vectorization possible but seems inefficient"**

Samples:

```
for (I=0;I<=MAX;I++)
  for (J=0;J<=MAX;J++)
  {
    D[I][J]+=1;                  // Unit Stride
    D[J][I]+=1;                  // Non-Unit but linear
    A[J*J]+=1;                   // Non-unit
    A[B[J]]+=1;                  // Non-Unit
    if (A[MAX-J])==1) last=J;    // Non-Unit
  }
```

Avoiding Non-Unit Stride Access

- Code transformations like loop interchange can avoid non-unit access frequently in case access is linear
- Compiler does this automatically in many cases; popular sample: matrix multiplication loop
  - The compiler will swap inner loops to get unit-stride access

```
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    for(k=0;k<N;k++)
      c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

- But in other cases, the exchange has to be done manually: The following loops are not interchanged implicitly:

```
// Non-unit access
for (j = 0; j < N; j++)
 for (i = 0; i <= j; i++)
  c[i][j] = a[i][j]+b[i][j];
```

```
// Unit access
for (i = 0; i < N; i++)
 for (j = i; i <= N; j++)
  c[i][j] = a[i][j]+b[i][j];
```

Function Calls / In-lining

- Function calls prevent vectorization in general
  - Exception #1 : Call of "intrinsic" functions like math routines
  - Exception #2 : Successful in-lining of called routine
    - Inter-procedural optimization enables in-lining of routines defined even in separate source files

Intel Compiler: 15 times faster by using –ipo (/Qipo on Windows) !*

```
for (i=1;i<nx;i++) {
     x = x0 + i*h;
     sumx = sumx + func(x,y,xp,yp);
}

float func(float x, float y, float xp, float yp)
{
  float denom;
  denom = (x-xp)*(x-xp) + (y-yp)*(y-yp);
  denom = 1./sqrt(denom);
  return denom;
}
```

*: Intel® C++ Compiler 12.0 U1 for Linux, Redhat Enterprise Linux 64bit 6.0, Intel XEON® X5560 processor, 2.8GHz

Function Calls / In-lining [2]

- Success of in-lining can be verified using the optimization report
  - {L&M}: `-opt-report -opt-report-phaseipo_inl`
  - {W}: `/opt-report /Qopt-report-phaseipo_inl`
- Intel compilers offer a large set of switches, directives and language extensions to control in-lining globally or locally
  - E.g `#pragma forceinline` which instructs the compiler to ignore the heuristic for in-lining and to inline all calls in the following statements/block (C/C++ only)
  - See compiler manual for details
- Inter-procedural optimization offers additional advantages to vectorization
  - Inter-procedural alignment analysis
  - Improved (more precise) dependence analysis

Vectorizable Mathematical Functions

Calls to most mathematical function in a loop body are "vectorized" too by calling vector versions of the function provided by the "Short Vector Math Library" – libsvml

- Libsvml is optimized for latency compared to the VML library component of Intel® MKL which realizes same functionality but which is optimized for throughput
- Routines in libsvml can be called explicitly too (see manual)

This is the set of mathematical routines which have a vector implementation in libsvml (Intel® Composer XE 2011)

acos	ceil	fabs	round
acosh	cos	floor	sin
asin	cosh	fmax	sinh
asinh	erf	fmin	sqrt
atan	erfc	log	tan
atan2	erfinv	log10	tanh
atanh	exp	log2	trunc
cbrt	exp2	pow	

Data Type Topics

- Objects (variables, constants, …) used in a statement to be vectorized may have different types and/or sizes
- The compiler frequently can still vectorize them using e.g. packed SSE/AVX conversion, insertion, extraction, …, instructions
- To analyze the cases, where this is not possible (“unsupported data type”), consider
  - the partially surprising rules for implicit data type promotions defined by the programming language standard
  - the potentially missing SSE/AVX instruction which would be needed here
  - The size differences for source and result operands potentially required for operations like a multiplication
- In case the complex data type (either single or double precision) is being used (Fortran, C99), Intel® SSE3 provides the basic arithmetic instructions to support vectorization

Student Exercise # 3
Understand Type Impact

Explain, why the code to the right vectorizes (SSE2) for
  SUM_TYPE == “int”
but does not for
  SUM_TYPE == “short”

Hints:
-what is the type v1 and v2 have to be promoted to before the multiplication ?
-Look at SSE2 instructions PMADDWD (multiply and add)

```
typedef int SUM_TYPE;

short ip(char *v1, char *v2)
{
  SUM_TYPE  inner_product  = 0;
  #pragma vector aligned

  for (int i=0; i<1024; ++i)
    inner_product += v1[i] * v2[i];

  return inner_product;
}
```

PMADDWD a, b: Mutiplies the 8 signed 16 bit integers from a by the 8 signed integers from b. Adds the signed 32bit integer results pairwise and packs the 4 signed 32-bit integer results into a

Control Flow/ Control Dependencies

- Control dependencies caused by a complex control flow within the loop body prevent vectorization in general
- However loops with "conditional statements" can be vectorized frequently using a bit masking technique
  - The idea outlined using a sample :

```
for (i=1; i<=U; i++)
  if ( R1[i] > R2[i] )
     L1[i] = R1[i];
  else
     L2[i] = R2[i];
```

```
MASK[1:U] = (R1[1:U] > R2[1:U]);
L1[1:U] = (MASK[1:U] & R1[1:U]) |
          (!MASK[1:U] & L1[1:U]);
L2[1:U] = (MASK[1:U] & L2[1:U]) |
          (!MASK[1:U] & R2[1:U]);
```

- This "if-conversion" by bit-masking works too for if-constructs with a "true" branch only
- The SSE instruction set facilitates a very compact and efficient construction of the bit mask generation and the masking operations
  - See sample code for SSE-intrinsics introduced earlier

Bit Masking – Guarding Errors

- This “if-conversion” causes both, the “if” and “else” part to be evaluated for all iterations. The compiler will do this only in case it can exclude to introduce errors which have been protected in the original code
  - Vector report: “… condition may protect exception”
  - In our sample, the compiler can be sure to not introduce a new exception since the right-end side expressions are touched in the test anyway already for each iteration
  - In some cases, the compiler will add tests before the loop to validate the correctness of the transformations
  - Vectorization-enabling directives like “#pragma vector always” will assure to compiler, that the masking transformation is safe

Student Exercise

a) Compile and run a simple program multiplying a matrix with a vector showing some of the topics introduced up to now like vectorization reports, dependence, memory disambiguation and alignment

b) A program showing the benefits of interprocedural optimization

Agenda

- Introduction
- Vector Code Generation
- Compiler Switches for Automatic Vectorization
- Validating Success of Automatic Vectorization
- When Vectorization fails
  - Data Dependence
  - Alignment
  - Others like Non-Unit Stride Access, Function Calls, …
- **Vectorization of special program constructs**
  - **Idiom Recognition**
  - **Complex data type**
- HLO Loop Transformations
- Summary, References

Idiom Recognition

The Intel compilers recognize program constructs which can be mapped onto compact idiomatic SSE/AVX instructions providing optimal performance

Sample:

```
unsigned char a[N], b[N];
void swap32(int n)
{
  int i;
  for (i = 0; i < n; i+=4)
  {
    a[i+0] = b[i+3];
    a[i+1] = b[i+2];
    a[i+2] = b[i+1];
    a[i+3] = b[i+0];
  }
}
```

⇨

```
L:  movdqa    xmm1, b[ecx*4-4]
    pshufb    xmm1, xmm0
    movdqa    a[ecx*4-4], xmm1
    add       ecx, 4
    cmp       ecx, eax
    jle       L
```

Byte swapping pattern is recognized as an idiom.
PSHUFB is an instruction introduced by SSSE-3; thus the corresponding transformation requires extension SSSE3 at least

Idiom Recognition – Saturation

- To enable idiom recognition, the source code needs to express exactly the conditions required to use the corresponding SSE instruction
- In the sample below, the compiler will use PADDSB (“Add packed signed bytes with saturation”) because the source code limits both, the upper and lower bound, of the add operation
  - Frequently the lower bound check is missing which would be ok here only for <u>unsigned char</u> !

```
define N 1000
void sat_signed_char(char va[N],char vb[N], char vc[N])
{
  int i;
  for (i = 0; i < N; i++)
    vc[i] = ( (vb[i] + va[i]  > 127 ) ? 127  :
                  ( ( vb[i] + va[i] < -128 ) ? -128 :
                                  vb[i] + va[i]  )    );
}
```


Optimization Notice

Vectorization for Complex Arithmetic

- Both C (C99) and Fortran provide explicit support for COMPLEX data type
- The Intel compilers can vectorize the corresponding arithmetic instructions using SSE3 instructions
- Sample for C :

```
float _Complex zc[10];

float _Complex za=4 + __I__*2;
      //Real Part            = 4
      //Imaginary part  = 2

void zscale() {
    for (int i=0; i<10; i++)
            zc[i] = za*csin(zc[i]);
}
//Compile (W): icl complex.c /Qstd=c99 /QxSSE3
//Compile (L & M): icl complex.c -std=c99 -xSSE3
```

Agenda

- Introduction
- Vector Code Generation
- Compiler Switches for Automatic Vectorization
- Validating Success of Automatic Vectorization
- When Vectorization fails
  - Data Dependence
  - Alignment
  - Others like Non-Unit Stride Access, Function Calls, …
- Vectorization of special program constructs
  - Idiom Recognition
  - Complex data type
- **HLO Loop Transformations**
- Summary, References

Loop Transformations

- Frequently (optimal) vectorization is possible only after adapting the loops before
- The compiler component responsible for these loop transformations is phase HLO – High Level Optimization
  - While HLO is active for optimization level O2 and O3, only O3 activates the full set of transformations and applies the transformations more aggressively
- Intel compilers provide detailed report on HLO activity:

  {L&M}: –opt-report –opt-report-phasehlo

  {W}: /Qopt-report /Qopt-report-phasehlo

```
…
LOOP INTERCHANGE in loops at line: 7 8 9
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )

…
Loop at line 7 unrolled and jammed by 4
Loop at line 8 unrolled and jammed by 4
…
```

Sample for Loop Transformations

```
14: for (i=0; i<100; i++)
15: {
16:   a[i] = 0;
17:   for (j=0; j<100; j++)
18:      a[i] += b[j][i];
19:  }
```

```
a[0:99] = 0;
for (j=0; j<100; j++)
  a[0:99] += b[j][0:99];
```

Report from vectorizer:
file.c(16) : (col. 8) remark: PARTIAL LOOP WAS VECTORIZED.
file.c(14) : (col. 8) remark: loop was not vectorized: not inner loop.
file.c(18) : (col. 10) remark: PERMUTED LOOP WAS VECTORIZED.

Transformations done by compiler:
1) i-loop is distributed into 2 loops: a single loop and a nested loop
2) Nested loop is interchanged to exploit spatial locality on b[j][i]
3) A single loop is vectorized. (1st VECTORIZED message)
4) Inner loop of the interchanged nested loop is vectorized (2nd VECTORIZED message)

Some HLO (Loop) Transformations
Enabled for –O3

– Loop interchange	(for more efficient memory access)
– Loop unrolling	(more instruction level parallelism)
– Cache blocking	(for more reuse of data in cache)
– Loop peeling	(allow for misalignment)
– Loop versioning	(for loop count, data alignment, …)
– Memcpy recognition	(call Intel’s fast memcpy, memset)
– Loop splitting	(facilitate vectorization)
– Loop fusion	(more efficient vectorization)
– Scalar expansion	(remove dependency)
– Loop rerolling	(enable vectorization)
– Loop reversal	(handle dependencies)

*Blue color: Applied too for optimization level O2

Student Exercise [optional]

a) Sample program showing benefit of Complex Arithmetic vectorization

b) Sample program for idiom recognition

Agenda

- Introduction
- Vector Code Generation
- Compiler Switches for Automatic Vectorization
- Validating Success of Automatic Vectorization
- When Vectorization fails
  - Data Dependence
  - Alignment
  - Others like Non-Unit Stride Access, Function Calls, …
- Vectorization of special program constructs
  - Idiom Recognition
  - Complex data type
- HLO Loop Transformations
- **Summary, References**

Summary

- Intel® C++ and Intel® Fortran Compilers of Intel® Composer XE provide sophisticated and flexible support for automatic vectorization
- Even for “automatic” vectorization explicit compilation support via developer can improve result considerably
  - Compiler provides reporting features to look at results
  - Directives and compiler switches permit fine-tuning for vectorization
- Some understanding of concepts like dependence and alignment is required to get best SSE/AVX performance out of the compilers

References

[1] Aart Bik: “The Software Vectorization Handbook”

– http://www.intel.com/intelpress/sum_vmmx.htm

[2] Intel® 64 and IA-32 Architectures Software Developer's Manuals

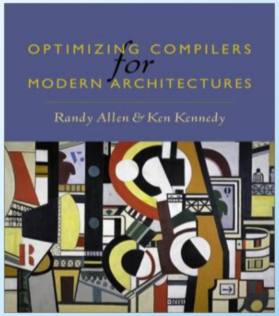
– http://www.intel.com/products/processor/manuals/index.htm

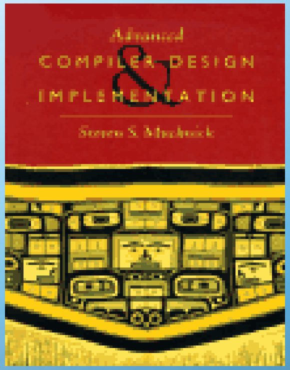
[3] Randy Allen, Ken Kennedy: “Optimizing Compilers for Modern Architectures: A Dependence-based Approach”

[4] Intel Software Forums, Knowledge Base, White Papers, Tools Support etc

– http://software.intel.com

[5] Steven S. Muchnik, “Advanced Compiler Design and Implementation”





Optimization Notice

Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel® Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101

Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

http://intel.com/software/products