Loop Independence, Compiler Vectorization and Threading of Loops (SSE & AVX)

Michael Klemm
Software & Services Group
Developer Relations Division

Legal Disclaimer

Optimization Notice

Agenda

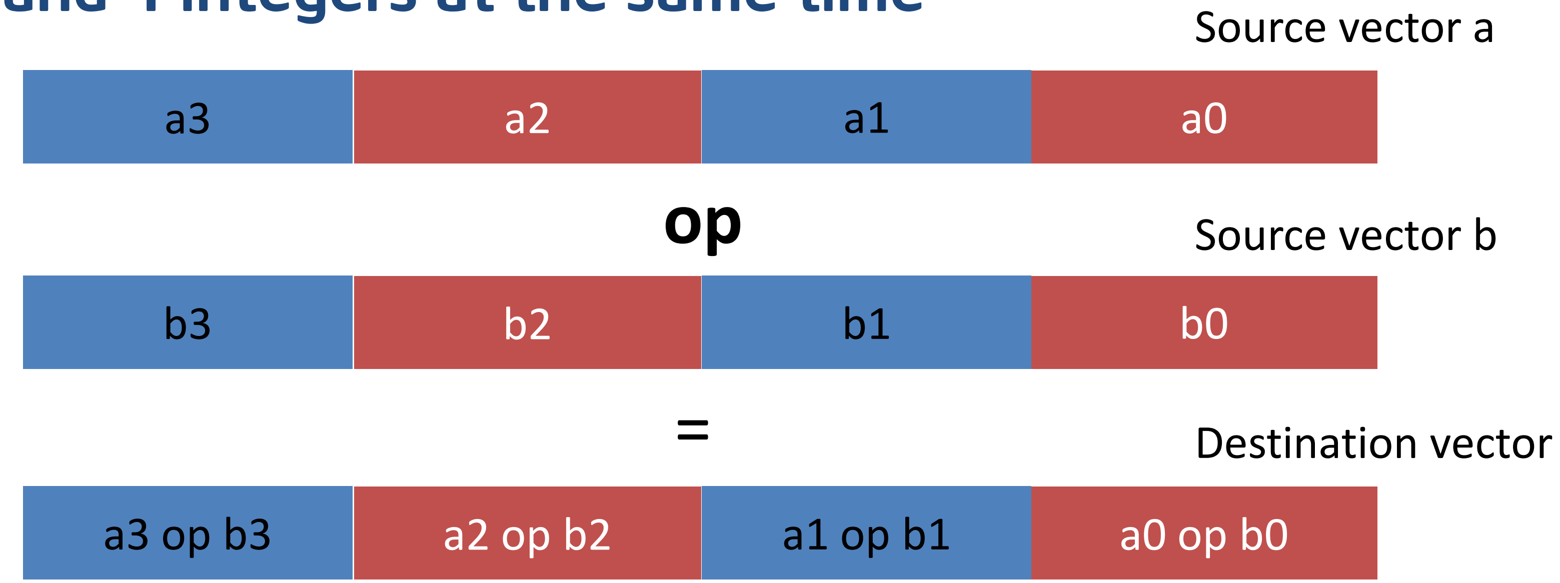
- Vector Instructions (SIMD)
- Compiler Switches for Optimization
- Controlling Auto-vectorization
- Controlling Auto-parallelization
- Manual Vectorization (SSE & AVX)
- Hands-on

Agenda

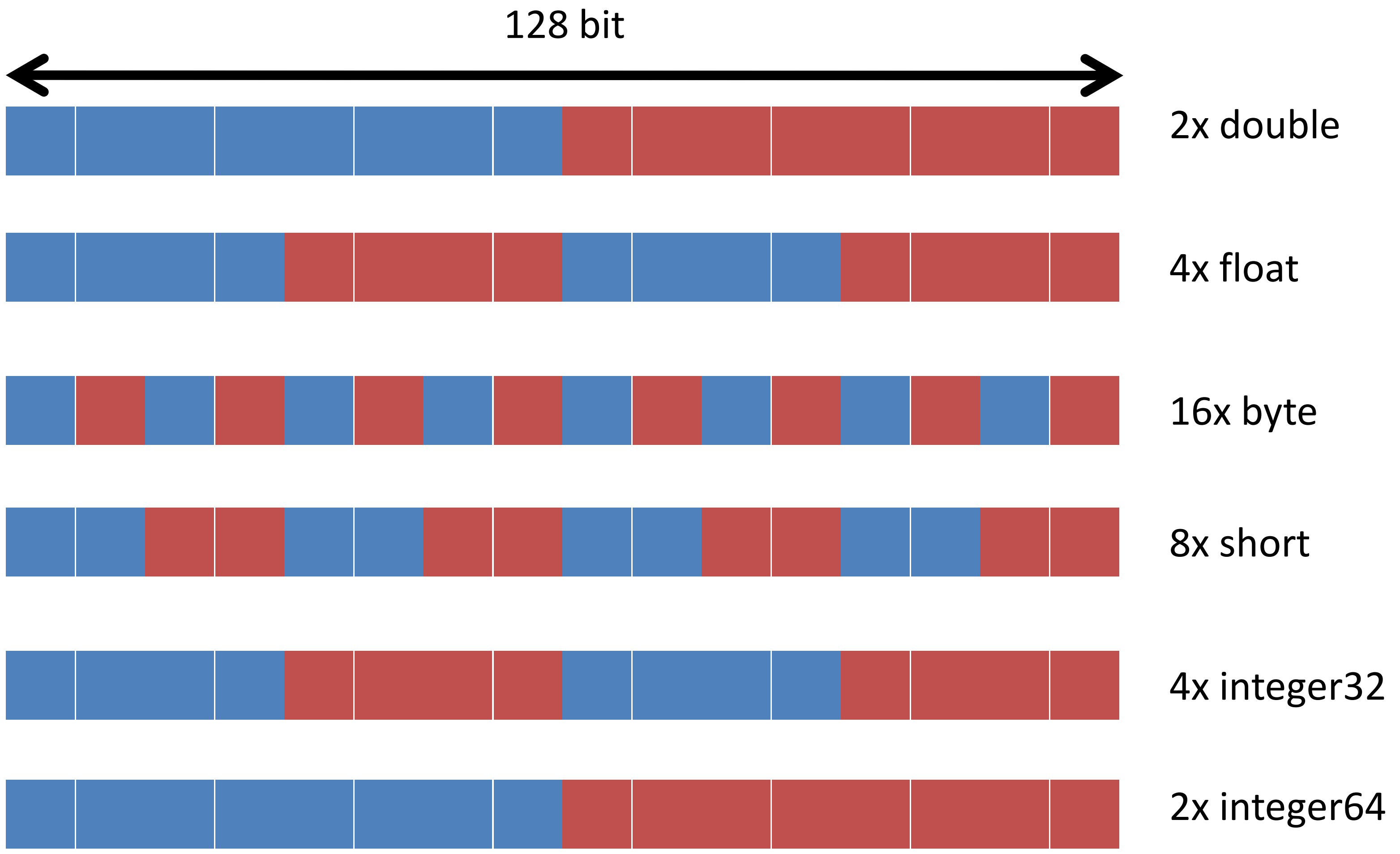
- Vector Instructions (SIMD)
- Compiler Switches for Optimization
- Controlling Auto-vectorization
- Controlling Auto-parallelization
- Manual Vectorization (SSE & AVX)
- Hands-on

What is SSE (and related instruction sets)

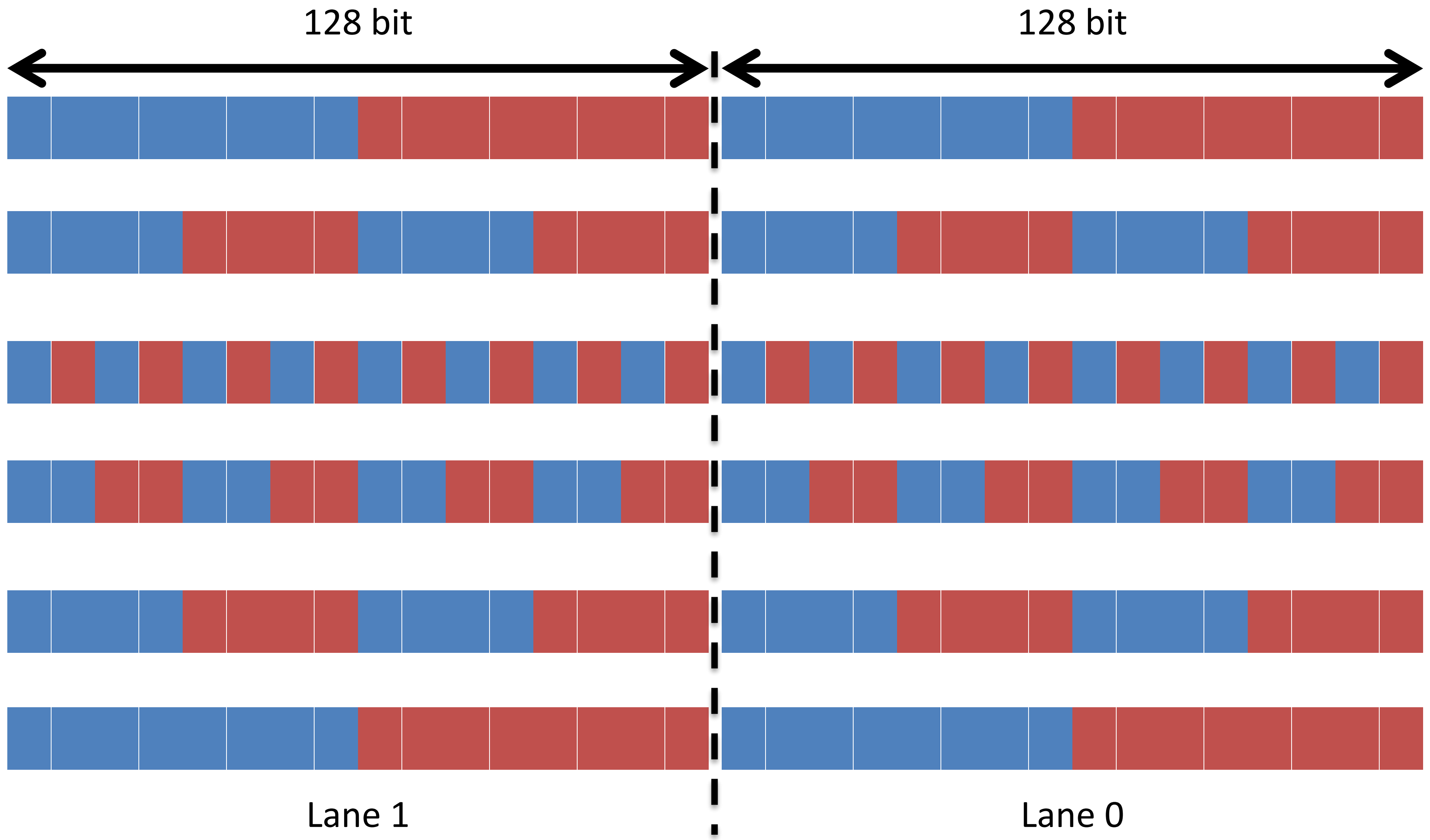
- SSE: Streaming SIMD extension
- SIMD: Single instruction, Multiple Data (Flynn's Taxonomy)
- SSE allows the identical treatment of 2 double, 4 floats and 4 integers at the same time



SSE Data Types



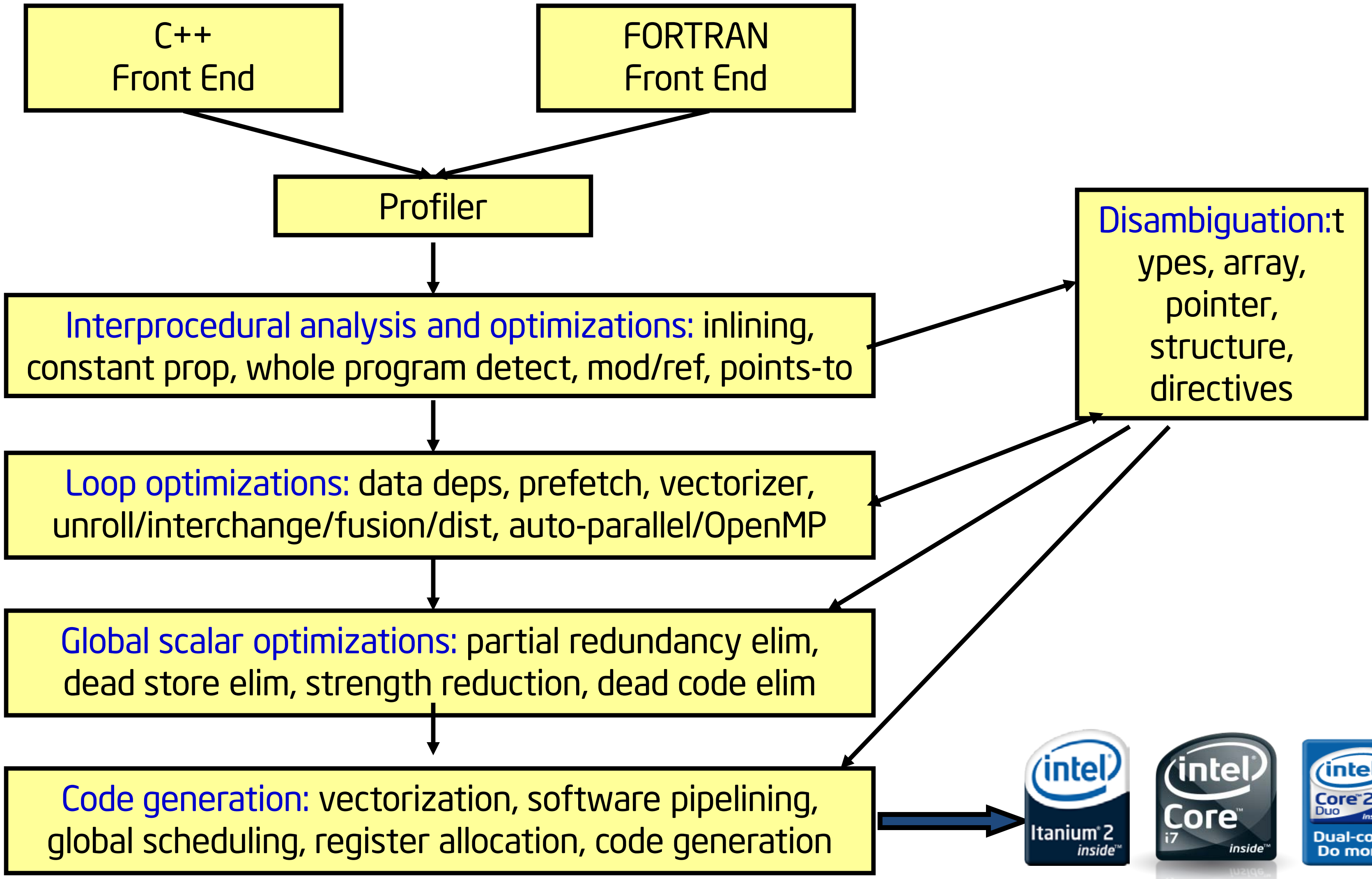
AVX Data Types



Agenda

- Vector Instructions (SIMD)
- Compiler Switches for Optimization
- Controlling Auto-vectorization
- Controlling Auto-parallelization
- Manual Vectorization (SSE & AVX)
- Hands-on

Intel® Compiler Architecture


C++ Front End
FORTRAN Front End
Profiler
Interprocedural analysis and optimizations: inlining, constant prop, whole program detect, mod/ref, points-to
Loop optimizations: data deps, prefetch, vectorizer, unroll/interchange/fusion/dist, auto-parallel/OpenMP
Global scalar optimizations: partial redundancy elim, dead store elim, strength reduction, dead code elim
Code generation: vectorization, software pipelining, global scheduling, register allocation, code generation
Disambiguation:t ypes, array, pointer, structure, directives

A few General Switches

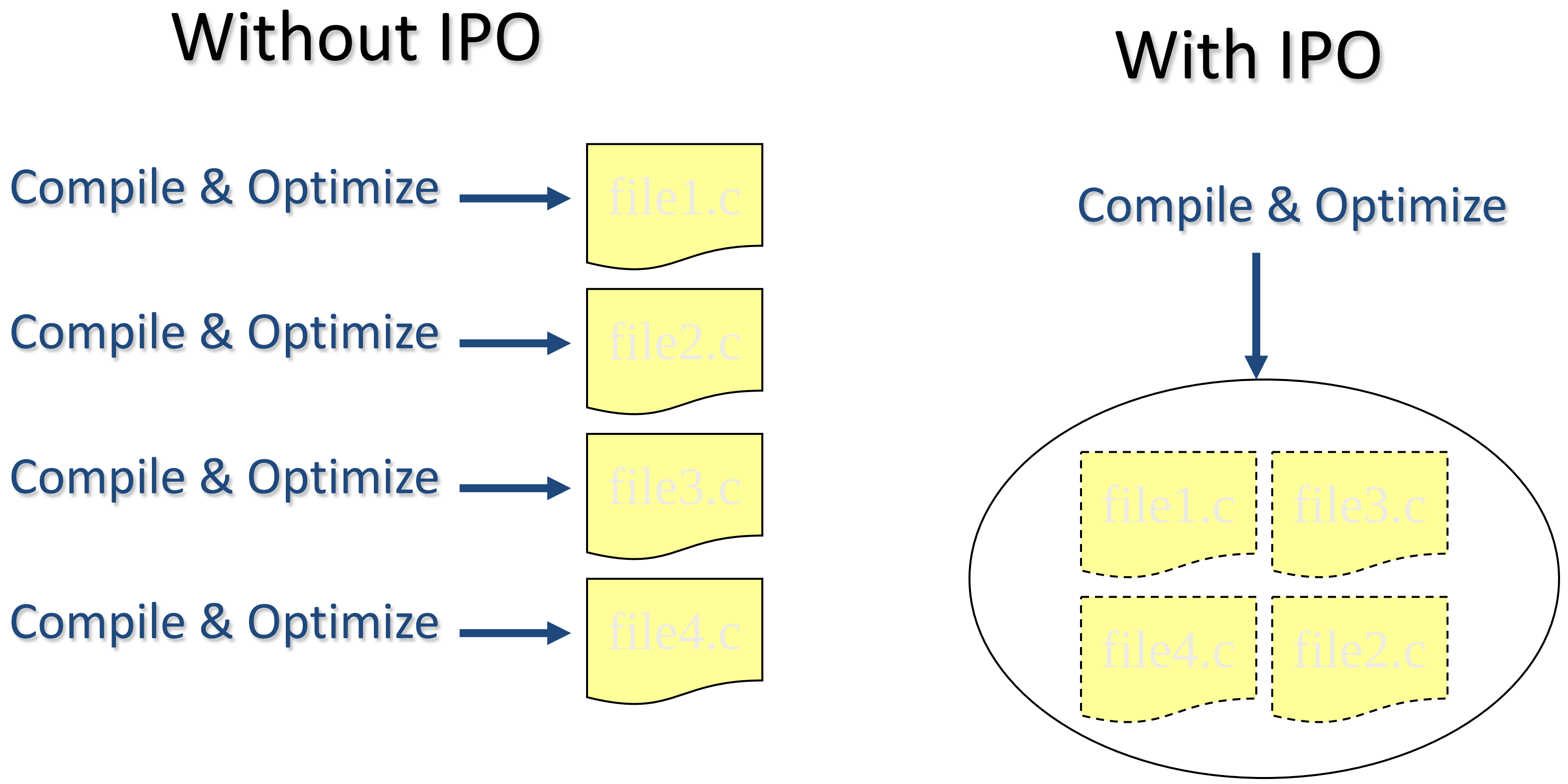
Functionality	Linux
Disable optimization	-O0
Optimize for speed (no code size increase), no SWP	-O1
Optimize for speed (default)	-O2
High-level optimizer (e.g. loop unroll), -ftz (for Itanium)	-O3
Vectorization for x86, -xSSE2 is default	<many options>
Aggressive optimizations (e.g. -ipo, -O3, -no-prec-div, -static -xHost for x86 Linux*)	-fast
Create symbols for debugging	-g
Generate assembly files	-S
Optimization report generation	-opt-report
OpenMP support	-openmp
Automatic parallelization for OpenMP* threading	-parallel

Architecture Specific Switches

Functionality	Linux
Optimize for current machine	-xHOST
Generate SSE v1 code	-xSSE1
Generate SSE v2 code (may also emit SSE v1 code)	-xSSE2
Generate SSE v3 code (may also emit SSE v1 and v2 code)	-xSSE3
Generate SSE v3 code for Atom-based processors	-xSSE_ATOM
Generate SSSE v3 code (may also emit SSE v1, v2, and v3 code)	-xSSSE3
Generate SSE4.1 code (may also emit (S)SSE v1, v2, and v3 code)	-xSSE4.1
Generate SSE4.2 code (may also emit (S)SSE v1, v2, v3, and v4 code)	-xSSE4.2
Generate AVX code	-xAVX

Interprocedural Optimization

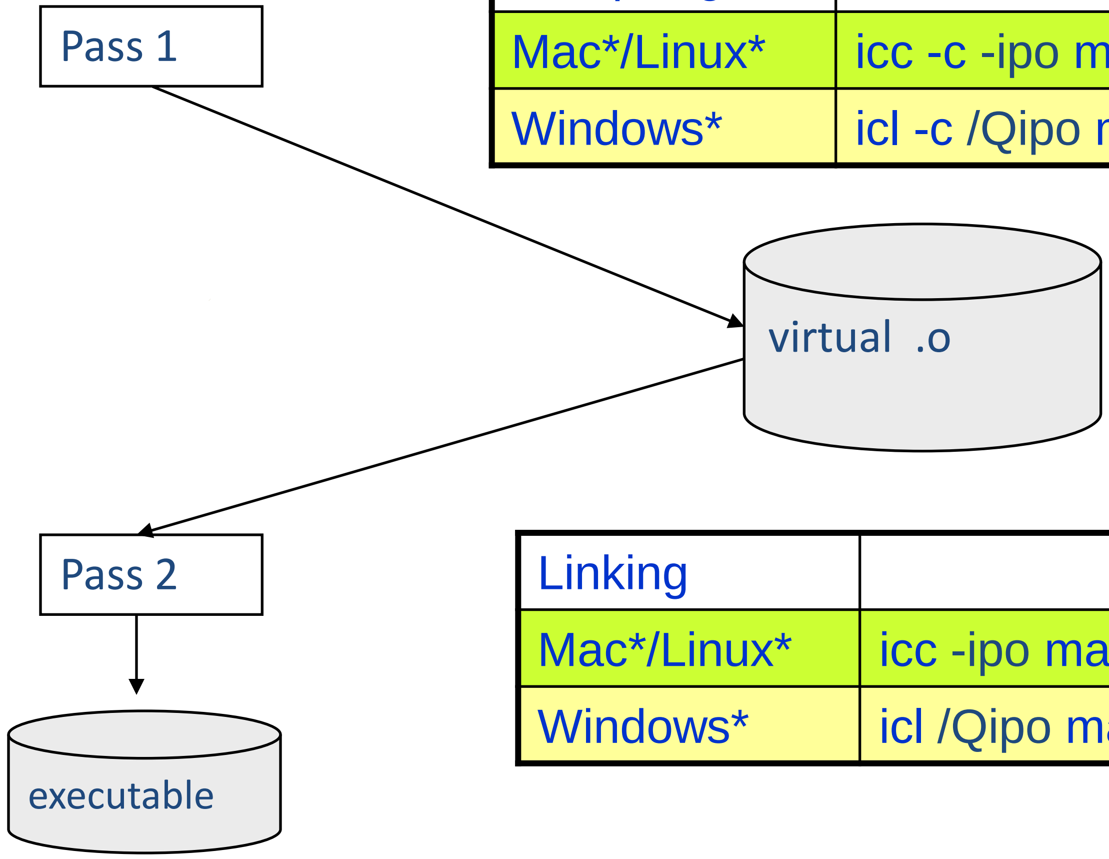
Extends optimizations across file boundaries



-ip	Only between modules of one source file
-ipo	Modules of multiple files/whole application

IPO – A Multi-pass Optimization

A Two-Step Process



Compiling	
Mac*/Linux*	icc -c -ipo main.c func1.c func2.c
Windows*	icl -c /Qipo main.c func1.c func2.c

Linking	
Mac*/Linux*	icc -ipo main.o func1.o func2.o
Windows*	icl /Qipo main.o func1.o func2.o

What you should know about IPO

- O2 and O3 activate “almost” file-local IPO (-ip)
  - Only a very few, time-consuming IP-optimizations are not done but for most codes, -ip is not adding anything
  - Switch –ip-no-inlining disables in-lining
- IPO extends compilation time and memory usage
  - See compiler manual when running into limitations
- Inlining of functions is the most important feature of IPO but there is much more
  - Inter-procedural constant propagation
  - MOD/REF analysis (for dependence analysis)
  - Routine attribute propagation
  - Dead code elimination
  - Induction variable recognition
  - …many, many more

Profile-Guided Optimizations (PGO)

- Use execution-time feedback to guide (final) optimization
- Helps I-cache, paging, branch-prediction
- Enabled optimizations:
  - Basic block ordering
  - Better register allocation
  - Better decision on which functions to inline
  - Function ordering
  - Switch-statement optimization

PGO Usage: Three Step Process

Step 1

Instrumented Compilation

icc -prof_gen prog.c

→ Instrumented executable: prog.exe

Step 2

Instrumented Execution

prog.exe (on a typical dataset)

→ DYN file containing dynamic info: .dyn

Step 3

Feedback Compilation

icc -prof_use prog.c

→ Merged DYN summary file: .dpi
Delete old dyn files unless you want their info included

Simple PGO Example: Code Re-Order

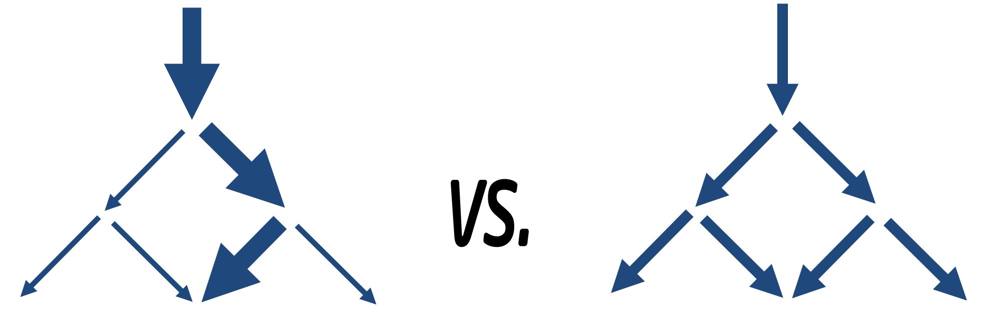
```
for (i=0; i < NUM_BLOCKS; i++)
{
    switch (check3(i))
    {
        case 3:                          /* 25% */
            x[i] = 3;  break;
        case 10:                         /* 75% */
            x[i] = i+10;  break;
        default:                         /* 0% */
            x[i] = 99;  break
    }
}
```

"Case 10" is moved to the beginning

– PGO can eliminate most tests&jumps for the common case – less branch mispredicts

What you should know about PGO

- Instrumentation run can be up to twice as long
  - In-lining disabled, trace calls overhead
- Sometimes trace-files cannot be found
  - Looking at right directory ?
  - Clean exit() call is necessary to dump info
    - Debugger can help / break in PGO trace start/end calls
- Benefit depends on control flow structure:



Significant Benefit

Little Benefit

Memory Reference Disambiguation

Options/Directives related to Aliasing

- -alias_args[-]
- -ansi_alias[-]
- -fno-alias: No aliasing in whole program
- -fno-fnalias: No aliasing within single units
- -restrict (C99): -restrict and *restrict* attribute
  - enables selective pointer disambiguation
- -safe_cray_ptr: No aliasing introduced by Cray-pointers
- -assume dummy_alias
- Related: Switch –ipo and directive IFDEP

Optimization Report Options

- opt_report
  - generate an optimization report to stderr (or file)
- opt_report_file <*file*>
  - specify the filename for the generated report
- opt_report_phase <phase_name>
  - specify the phase that reports are generated against
- opt_report_routine <*name*>
  - reports on routines containing the given name
- opt_report_help
  - display the optimization phases available for reporting
- vec-report<level>
  - Generate vectorization report (IA32, EM64T)

What did the Compiler do ?

ipo	ilo	**hpo**
ipo_inl	ilo_lowering	hpo_analysis
ipo_cp	**ilo_strength_reduction**	**hpo_openmp**
ipo_align	ilo_reassociation	**hpo_vectorization**
ipo_modref	ilo_copy_propagation	**hpo_threadization**
ipo_lpt	ilo_convert_insertion	
ipo_subst	ilo_convert_removal	**ecg**
ipo_vaddr		ecg_code_cycles
ipo_psplit	**hlo**	ecg_code_size
ipo_gprel	hlo_scalar_expansion	ecg_predication
ipo_pmerge	**hlo_unroll**	ecg_profiling
ipo_fps	**hlo_distribution**	ecg_inl
ipo_ppi	**hlo_fusion**	ecg_copy_propagation
ipo_unref	**hlo_prefetch**	ecg_brh
ipo_wp	hlo_loop_collapsing	ecg_speculation
		ecg_swp
	hlo_reroll	
pgo	hlo_loadpair	(ecg*, loadpair for IPF only)

List of most phases the user can ask to get detailed reports from

Only a few phases are relevant for the typical compiler user but these are really helpful !
Make use of it – much easier than assembler code inspection

Sample HLO Report

icc -O3 -opt_report -opt_report_phase hlo

```
…
LOOP INTERCHANGE in loops at line: 7 8 9
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )
LOOP INTERCHANGE in loops at line: 15 17
Loopnest permutation ( 1 2 3 ) --> ( 3 2 1 )
…

Loop at line 7 unrolled and jammed by 4
Loop at line 8 unrolled and jammed by 4
Loop at line 15 unrolled and jammed by 4
Loop at line 16 unrolled and jammed by 4
…
```

Some Intel-Specific Compiler Directives

#pragma	Architecture	Description
vector/novector	IA-32, Intel64	Indicates to the compiler that the loop should (not) be vectorized overriding the compiler's heuristics
loop count(n)	IA-32, Intel64, IA-64	Place before a loop to communicate the approximate number of iterations the loop will execute. Affects software pipelining, vectorization and other loop transformations.
distribute point	IA-32, Intel64, IA-64	Placed before a loop, the compiler will attempt to distribute the loop based on its internal heuristic. Placed within a loop, the compiler will attempt to distribute the loop at the point of the pragma. All loop-carried dependencies will be ignored.
unroll, unroll(n), nounroll	IA-32, Intel64, IA-64	Place before an inner loop (ignored on non-inmost loops). #pragma unroll without a count allows the compiler to determine the unroll factor. #pragma unroll(n) tell the compiler to unroll the loop n times. #pragma nounroll is the same as #pragma unroll(0).
loop count n loop count n1, n2, n3 … loop count min=<l>, avg=<a>, max=<u>	IA-32, Intel64, IA-64	Hint to the compile on expected loop iteration count: (n): always n (n1, n2, n3 …) either n1, n2, n3 … expected minimum count <l>, average count <a> and maximal count <u>
ivdep	IA-32, Intel64, IA-64	Place before a loop to control vectorizaton/software pipelining The compiler is instructed to ignore "assumed" (not proven) dependencies preventing vectorization/software pipelining. For Itanium: Assume no BACKWARD dependencies, FORWARD loop-carried dependencies still can exist w/o preventing SWP. Use with –ivdep_parallel option to exclude loop-carried dependencies completely (e.g. for indirect addressing)

C/C++ Compiler for Linux*

- Main goals/values of Intel C/C++ Linux Compiler:
  - Compatible to GCC
  - In general much faster than GCC
    - In particular for typical HPC applications
  - Provide critical features not offered by GCC like support for latest Intel® processors
- Compatibility splits into 3 parts:
  - C/C++ source code language acceptance
    - Almost done; missing features are questionable (e.g nested function in C supported by GCC)
  - Switch-compatibility
    - Achieved for most relevant options
  - Object Code interoperability
    - 100% reached today – even for complex C++ and OpenMP*
- The ultimate compatibility test: Linux kernel build
  - No manual changes to source code required but requires 'wrapper' script

Agenda

- Vector Instructions (SIMD)
- Compiler Switches for Optimization
- Controlling Auto-vectorization
- Controlling Auto-parallelization
- Manual Vectorization (SSE & AVX)
- Hands-on

Auto-vectorization

- Key requirements
  - A compiler must not alter the program semantics
  - If the compiler cannot determine all dependencies, it has to forego vectorization

- Compilers sometimes need to act very conservatively
  - Pointers make it hard for the compiler to deduce memory layout
  - Codes may produce overlapping arrays through pointer arithmetics
  - If the compiler can't tell, it does not vectorize

Data Dependencies

- Suppose two statements S1 and S2
- S2 depends on S1, iff S1 must be executed before S2
  - Control-flow dependence
  - Data dependence
  - Dependencies can be carried over between loop iterations
- Flavors of data dependencies

FLOW

$$\begin{tikzcd}[row sep=small] \text{s1: } a = 40 \arrow[dd, bend left] \\ \quad b = 21 \\ \text{s2: } c = a + 2 \end{tikzcd}$$

ANTI

$$\begin{tikzcd}[row sep=small] \quad b = 40 \\ \text{s1: } a = b + 1 \\ \text{s2: } b = 21 \arrow[u, bend right] \end{tikzcd}$$

Compiler Options for Vectorization

- Vectorization is automatically enabled for O2
- Can be controlled through command line switches:

-vec	enable vectorization
-no-vec	disable vectorization

- Vectorization reports help find out about (non-)vectorized code

-vec-report*n* enable vectorization diagnostics

0	report no diagnostic information.
1	report on vectorized loops (default)
2	report on vectorized and non-vectorized loops.
3	report on vectorized and non-vectorized loops and any proven or assumed data dependences.
4	report on non-vectorized loops.
5	report on non-vectorized loops and the reason why they were not vectorized.

```
source /opt/intel/Compiler/11.1/064/bin/iccvarsh.sh intel64
```

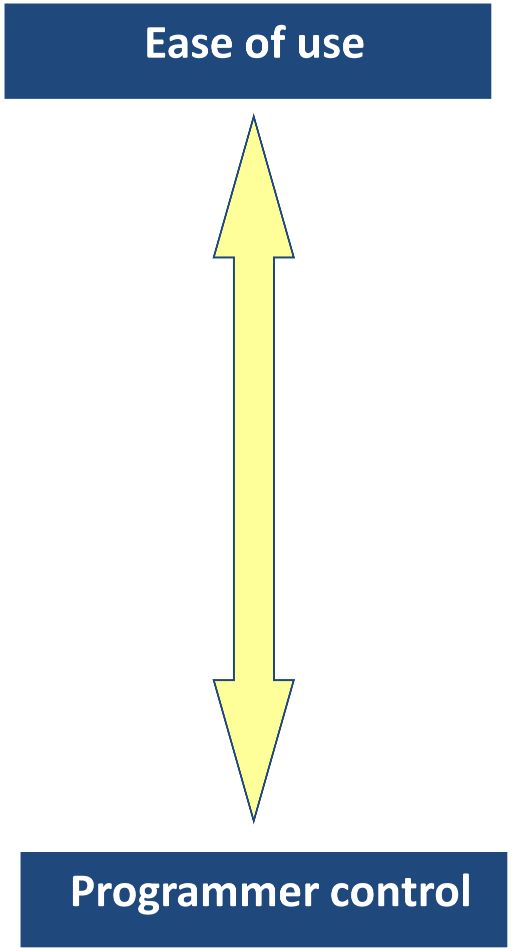
Vectorization Hints

- Sometimes, the compiler needs some help when looking at your code
- Compiler pragmas:
  - #pragma ivdep
    Indicate that there is no loop-carried dependence in the loop
  - #pragma vector always | aligned | unaligned
    Compiler is instructed to always vectorize a loop (and ignore internal heuristics)

• always:	always vectorize
• aligned:	use aligned load/store instructions
• anligned:	use unaligned load/store instructions

Position of SIMD Features

- Fully automatic vectorization
- Auto vectorization hints (#pragma ivdep)
- SIMD feature (#pragma simd and simd function annotation)
- SIMD intrinsic class (F32vec4 add)
- Vector intrinsic (mm_add_ps())
- ASM code (addps)



Why Didn't My Loop Vectorize?

- **Linux** `-vec-reportn` **Windows** `/Qvec-reportn`

- Set diagnostic level dumped to stdout

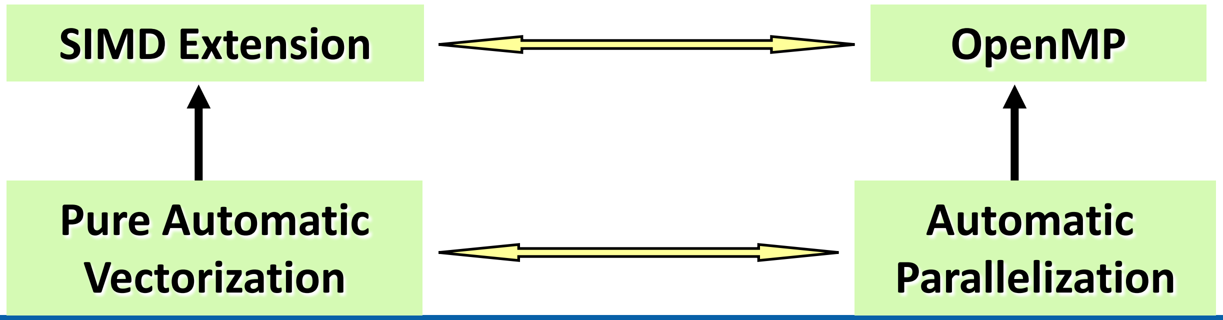
  n=0: No diagnostic information
  n=1: (Default) Loops successfully vectorized
  n=2: Loops not vectorized – and the reason why not
  n=3: Adds dependency Information
  n=4: Reports only non-vectorized loops
  n=5: Reports only non-vectorized loops and adds dependency info

SIMD Extension for Vector-Level Parallelism

User-mandated vectorization using new SIMD Pragma and SIMD Function Annotation

- SIMD pragma provides additional information to compiler to enable vectorization of loops (at this time only inner loop)
- SIMD Function Annotations add inter-procedural information to facilitate vectorization
- Supplements automatic vectorization but differently to what traditional pragmas ("automatic vectorization hints") like IVDEP, VECTOR ALWAYS, etc do
  - Traditional pragmas: A hint; not necessary overriding compiler's heuristic
  - New SIMD extensions : More like an assertion: in case vectorization still fails, it is considered a fault (an option controls whether it is really treated as error);

•Relationship similar to OpenMP versus automatic parallelization



SIMD Directives: Sample

```
foo(float *a, float *b, float *c, int n)
{
  for (int k=0; k<n; k++)   c[k] = a[k] + b[k];
}
```

Due to the overlapping nature of array accesses from the different call sites, it might not be semantically correct to use restrict keyword or IVDEP directive (there are dependencies between iterations for one call)

But it might be true for all calls, that e.g 4 consecutive iterations can be executed in parallel without violating any dependencies

```
void foo(float *a, float *b, float *c, int n)
{
  #pragma simd vectorlength(4)
  for (int k=0; k<n; k++ )  c[k] = a[k] + b[k];
}
```

SIMD Directive and Clauses

#pragma simd [<clause-list>]

- No clause – that is : #pragma simd
  - Enforce vectorization ; ignore dependencies etc
- vectorlength(n_1, n_2, …, n_N)
  - Executing n_i iterations as one vector instruction of vector length n_i, is semantically equivalent to n_i scalar interations
- private(var_1, var_2, …, var_N)
  - variables private to each iteration. initial value is broadcast to all private instances, and the last value is copied out from the last iteration instance.
- linear(var_1:$step_1$, var_2:$step_2$, …, var_N:$step_N$)
  - for every iteration of scalar loop, var_i is incremented by $step_i$,. Every iteration of the vector loop, therefore increments var_i by VL*$step_i$
- reduction(operator:var_1, var_2,…, var_N)
  - Loop code implements reduction (like "+") on arguments listed which can be vectorized
- [no]assert
  - to assert or not to assert when the vectorization fails. Default is to assert for SIMD pragma.

SIMD Function Annotation: Sample

```
__attribute__((simd)) float foo(float);

void vfoo(float *restrict a, float *restrict b, int n) {
  int i;
  for (i=0; i<n; i++)   a[i] = foo(b[i]);
}

float foo(float x) {
  ...
}

$ icc example.c -O3 -vec-report3 –restrict –vec:simd

example.c(9): (col. 3) remark: LOOP WAS VECTORIZED.
example.c(14): (col. 3) remark: FUNCTION WAS VECTORIZED
```

The SIMD annotation introduces 'foo' as a function which will be available as a vectorized version. For the calling functions this enables vectorization of the loop and will adapt argument and return value handling correspondingly

SIMD Function Annotations

(Only a Subset)

__attribute__((simd [(*clauses*)])) and clause can be

- processor(cpuid)
  - Generate a vector version of the function for the given processor. The default processor is taken from the implicit or explicit processor- or architecture- specific flag in the compiler command line.
- linear($param_1$:$step_1$, $param_2$:$step_2$, …,$param_N$:$step_N$)
  - Consecutive invocations of the function increment $param_i$ by $step_i$
- scalar($param_1$, $param_2$, …, $param_N$)
  - indicates the values of these parameters can be broadcasted to all iterations as a performance optimization
- mask
  - to generate a masked vector version of the function.
- user
  - not to generate this vector version since it is provided by the programmer.

CEAN: C/C++ Extensions for Array Notation

Add array notation to C/C++ similar to what was done for Fortran language in Fortran90 to express data parallel (e.g. SSE-) code explicitly

Sample:

```
// Traditional:
for (i=0; i < M-K; i++)
{
    s = 0;
    for (j = 0; j < K; j++)
        s += x[i+j] * c[j]
    y[i] = s;
}
```

```
// CEAN Version – outer loop
// “vectorized”

y[0:M-K] = 0;

for (j = 0; j < K; j++)
    y[0:M-K]+= x[j:M-K]*c[j];
```

- Nothing totally new; many similar initiatives in the past 30 years
  - E.g.: “Vector C” for Control Data 205, +25 years ago
- CEAN requires switch –farray-notation in 12.0 Compiler

CEAN Array Sections

- Array sections
  - Specification per dimension is lower bound : count [:stride]
  - This is different from Fortran lower bound : upper bound : [stride]
  - Samples:

```
A[:]      // All of vector A
B[2:6]    // Elements 2 to 7 of vector B
C[:][5]   // Column 5 of matrix C
D[0:3:2] // Elements 0,2,4 of vector D
```

- Basic, data parallel operations on array sections
  - Support for most C/C++ arithmetic and logic operators like “+”, “/”, ”<“, “&&”, “+=“, …
  - The shape of the sections must be identical, scalars are expanded implicitly
  - Sample:

```
( a[0:s]+b[5:s] )* pi  // pi * {a[i]+b[i+5], (i=0;i<s;i++)}
```

CEAN Assignment

Assignment evaluates LHS and assigns values to RHS in parallel

- The shapes of the RHS and LHS array section must be the same
- Scalar is expanded automatically
- Conceptually, RHS is evaluated completely before LHS
  - Compiler ensures semantic in code being generated
  - This can be a critical performance issue (temporary array variable)
- Samples:

```
a[:][:] = b[:][2][:] + c;
e[:] = d;            // scalar expansion
e[:] = b[:][1][:];   // error, shapes different
a[:][:] = e[:];      // error, shapes different
a[b[0:s]] = c[:]     // scatter operation
c[0:s] = a[b[:]]     // gather operation
```

Software & Services Group

CEAN – Some Advanced Features

- Functions can take array sections as arguments and can return sections
  - Any assumption on order (side effects) are a mistake
  - Compiler may generate calls to vectorized library functions
  - Examples:

```
a[:] = pow(b[:], c);   // b[:]**c
a[:] = pow(c, b[:]);   // c**b[:]
a[:] = foo(b[:])       // user defined
```

- Reductions combine elements in an array section into a single value using pre-defined operators or a user function
  - Pre-defined, e.g. _sec_reduce_{add, mul, min, …}

```
sum = __sec_reduce_add(a[:]*b[:]);    // dot product
res = __sec_reduce(fn, a[:], 0);      // apply function fn
```

Agenda

- Vector Instructions (SIMD)
- Compiler Switches for Optimization
- Controlling Auto-vectorization
- Controlling Auto-parallelization
- Manual Vectorization (SSE & AVX)
- Hands-on

Auto-parallelization

- Key requirements
  - A compiler must not alter the program semantics
  - If the compiler cannot determine all dependencies, it has to forego parallelization

- Compilers sometimes need to act very conservatively
  - Pointers make it hard for the compiler to deduce memory layout
  - Codes may produce overlapping arrays through pointer arithmetics
  - If the compiler can't tell, it does not parallelize

- Past 30 years have shown that auto-parallelization
  - is a tough problem in general
  - is only applicable to very regular loops
  - cannot take care of manual parallelization tasks

Compiler Options for Parallelization

- Vectorization is not automatically enabled
- Can be controlled through command line switches:

-parallel	enable parallelization	
-par-threshold*n*	parallelization threshold	
	n = 0	parallelize always
	n = 100	parallelize only if performance gain is 100%
	n = 50	parallelize if probability of performance gain is 50%

- Parallelization reports:

  -par-report*n* enable vectorization diagnostics

0	report no diagnostic information.
1	report on successfully parallelized loops (default)
2	report on successfully and unsuccessfully parallelized loops
3	like 2, but also give information about proven and assumed data dependendies

Compiler Options for Parallelization

- Controlling scheduling: -par-schedule-*keyword*

auto	Lets the compiler or run-time system determine the scheduling algorithm.
static	Divides iterations into contiguous pieces.
static-balanced	Divides iterations into even-sized chunks.
static-steal	Divides iterations into even-sized chunks, but allows threads to steal parts of chunks from neighboring threads.
dynamic	Gets a set of iterations dynamically.
guided	Specifies a minimum number of iterations.
guided-analytical	Divides iterations by using exponential distribution or dynamic distribution.
runtime	Defers the scheduling decision until run time.

Agenda

- Vector Instructions (SIMD)
- Compiler Switches for Optimization
- Controlling Auto-vectorization
- Controlling Auto-parallelization
- Manual Vectorization (SSE & AVX)
- Hands-on

Element-wise Vector Multiplication

```
void vec_eltwise_product(vec_t* a, vec_t* b,
                         vec_t* c) {
    size_t i;
    for (i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}
```

Element-wise Vector Multiplication (SSE)

```
void vec_eltwise_product_sse(vec_t* a, vec_t* b,
                             vec_t* c) {
    size_t i;
    __m128 va;
    __m128 vb;
    __m128 vc;
    for (i = 0; i < a->size; i += 4) {
        va = _mm_loadu_ps(&a->data[i]);
        vb = _mm_loadu_ps(&b->data[i]);
        vc = _mm_mul_ps(va, vb);
        _mm_storeu_ps(&c->data[i], vc);
    }
}
```

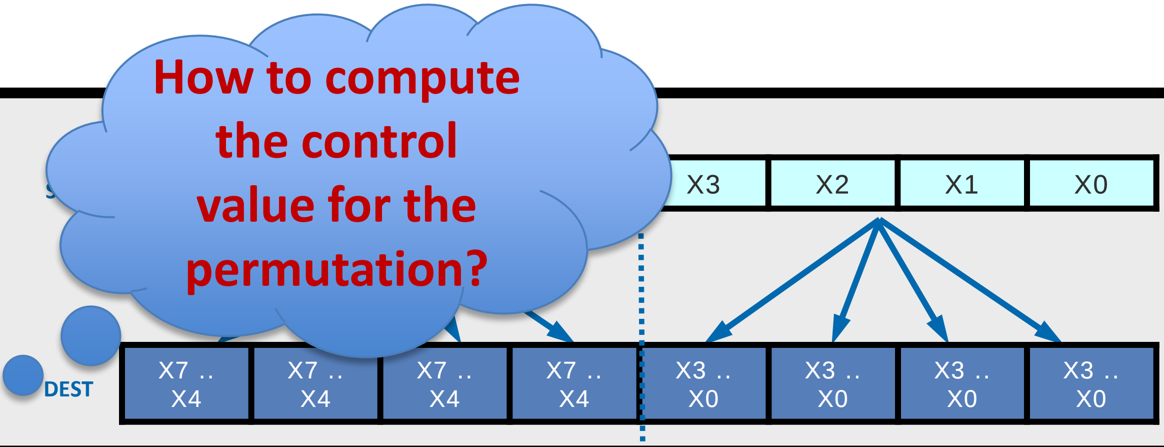
Element-wise Vector Multiplication (SSE)

```
void vec_eltwise_product_avx(vec_t* a, vec_t* b,
                             vec_t* c) {
    size_t i;
    __m256 va;
    __m256 vb;
    __m256 vc;
    for (i = 0; i < a->size; i += 8) {
        va = _mm256_loadu_ps(&a->data[i]);
        vb = _mm256_loadu_ps(&b->data[i]);
        vc = _mm256_mul_ps(va, vb);
        _mm256_storeu_ps(&c->data[i], vc);
    }
}
```

Zen and the Art of Permutation

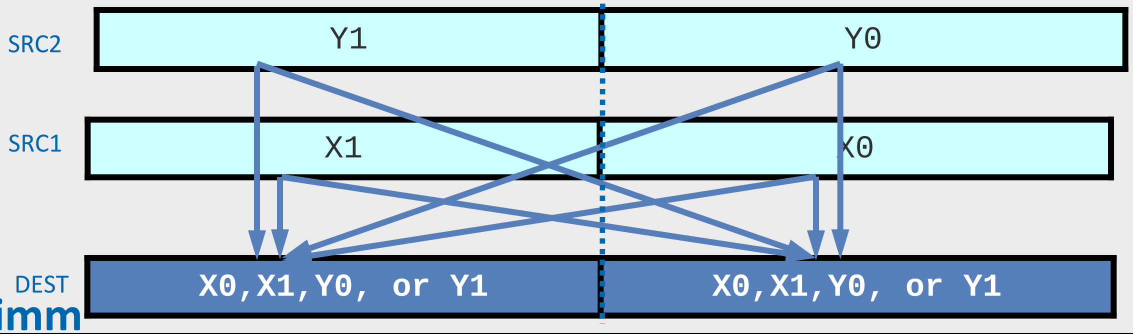
- **New in-lane PS and PD Permutes**
  - Permute controlled via immediate

vPermilPS dest, src, imm

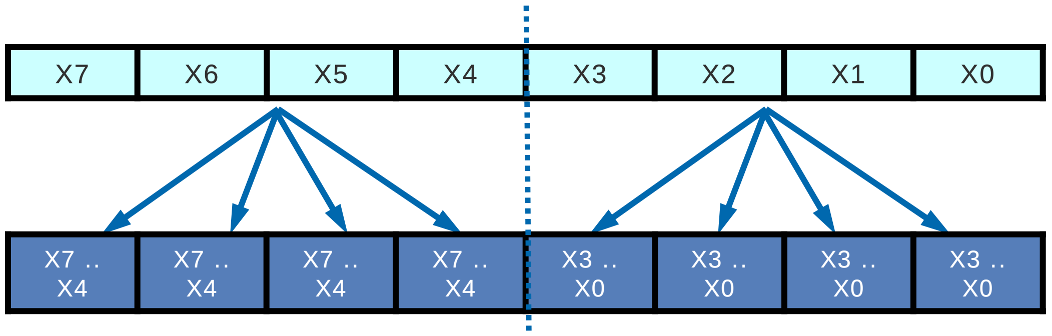


- **New 128-bit permutes**
  - Useful for lane-crossing operations

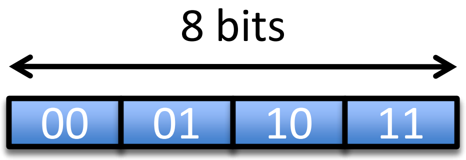
vPerm2F128 dest, src1, src2, imm



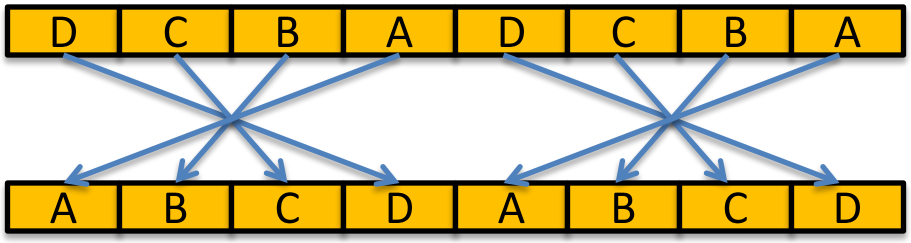
Zen and the Art of Permutation



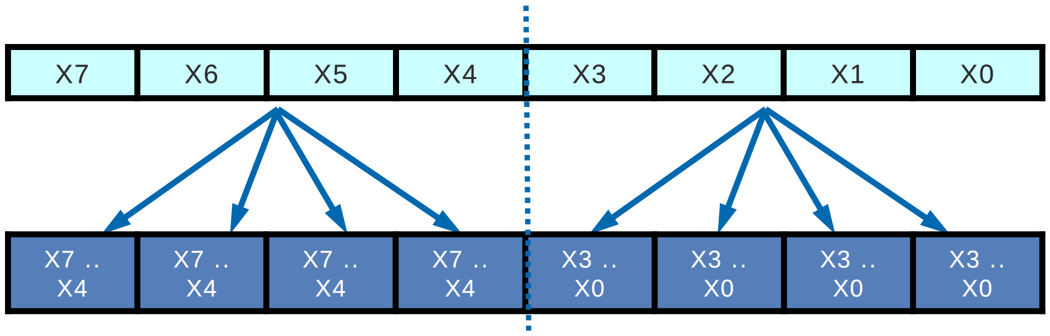
Control value:



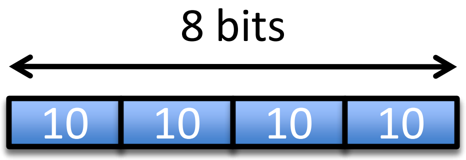
= 27 (0x1B)



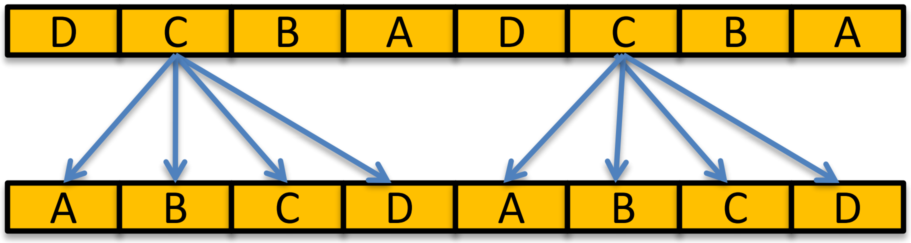
Zen and the Art of Permutation



Control value:



= 170 (0xAA)



Agenda

- Vector Instructions (SIMD)
- Compiler Switches for Optimization
- Controlling Auto-vectorization
- Controlling Auto-parallelization
- Manual Vectorization (SSE & AVX)
- Hands-on

Lab 1: Vectorization

- Look into the "vectors" directory.
- There you will find a code like this:

```
void scalar_product(vec_t* a, vec_t* b, vec_t *c) {
    size_t i;
    for (i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}
```

- Try to find out
  - Why the compiler cannot vectorize the code
  - Do you need to change the code?
  - Which compiler options do you need to enable vectorization

Lab 2: Parallelization

- Look into the “vectors” directory.
- There you will find a code like this:

```
void scalar_product(vec_t* a, vec_t* b, vec_t *c) {
    size_t i;
    for (i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}
```

- Try to find out
  - Why the compiler cannot parallelize the code
  - Do you need to change the code?
  - Which compiler options do you need to enable parallelization

Lab 3: Manual Vectorization

- Look into the “vectors” directory.
- Implement the SSE and AVX functions for computing the scalar product of two vectors.
  - The functions are marked with “TODO”.
- Hints:
  - You will need the following new instructions:
    SSE: _mm_hadd_ps()
    AVX: _mm256_maskstore_ps
- The “Intrinsics Guide for Intel® AVX” is very useful for the hands-on (see http://software.intel.com/en-us/avx/)

Questions?

intel®

Software