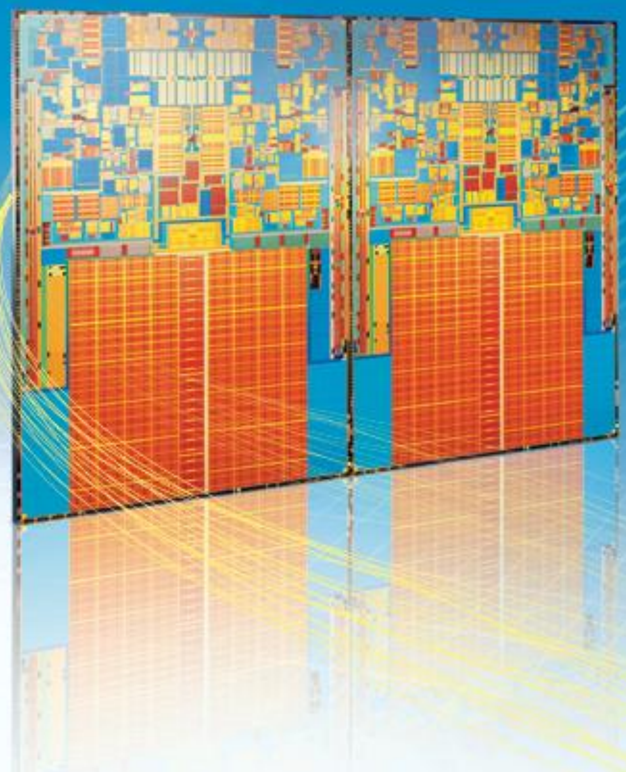
Vector Programming
A proposal for WG21

(presented to CPLEX within WG14)



N3734

2013-09-02

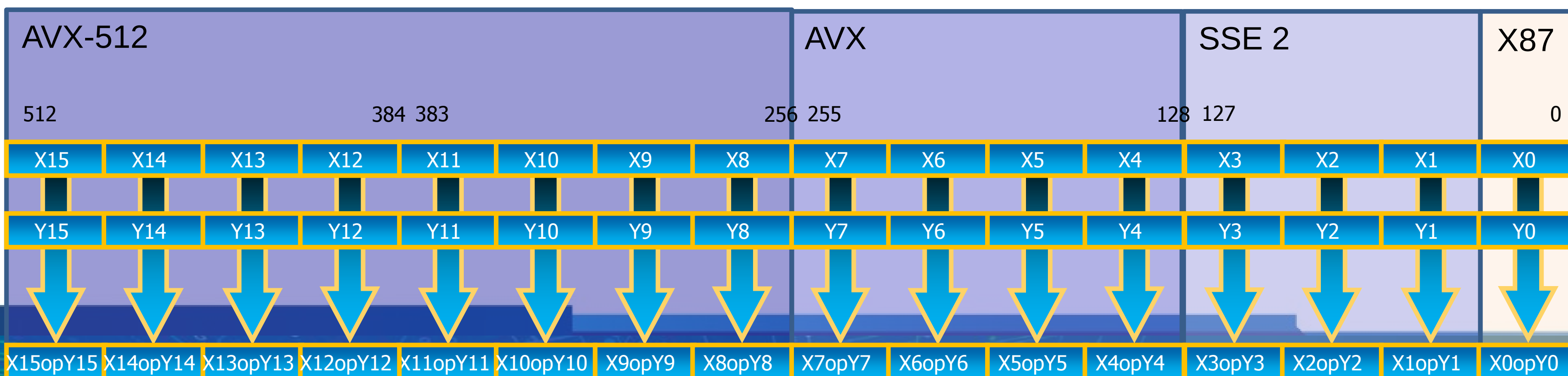
Robert Geva

Context

- Posing a presentation rather than a document
  - Multiple documents with the syntax and semantics shows in SG1 to date
- In Bristol, semantics were presented w/o a way for the programmer to actualize them
- This is the next step, presenting syntax
- The presentation covers vector loops, elemental functions and array notation.
  - Array notations may not be presented in the Chicago meeting.
- A comment on the dual syntax:
  - The syntax that is actually implemented and shipping is the #pragma based syntax.
  - The syntax being proposed is the keyword based syntax
  - The expectation is that capabilities, semantics and performance are the same
  - Some examples and illustrations here use #pragma, some use keywords.

SIMD / Vector Hardware Resources

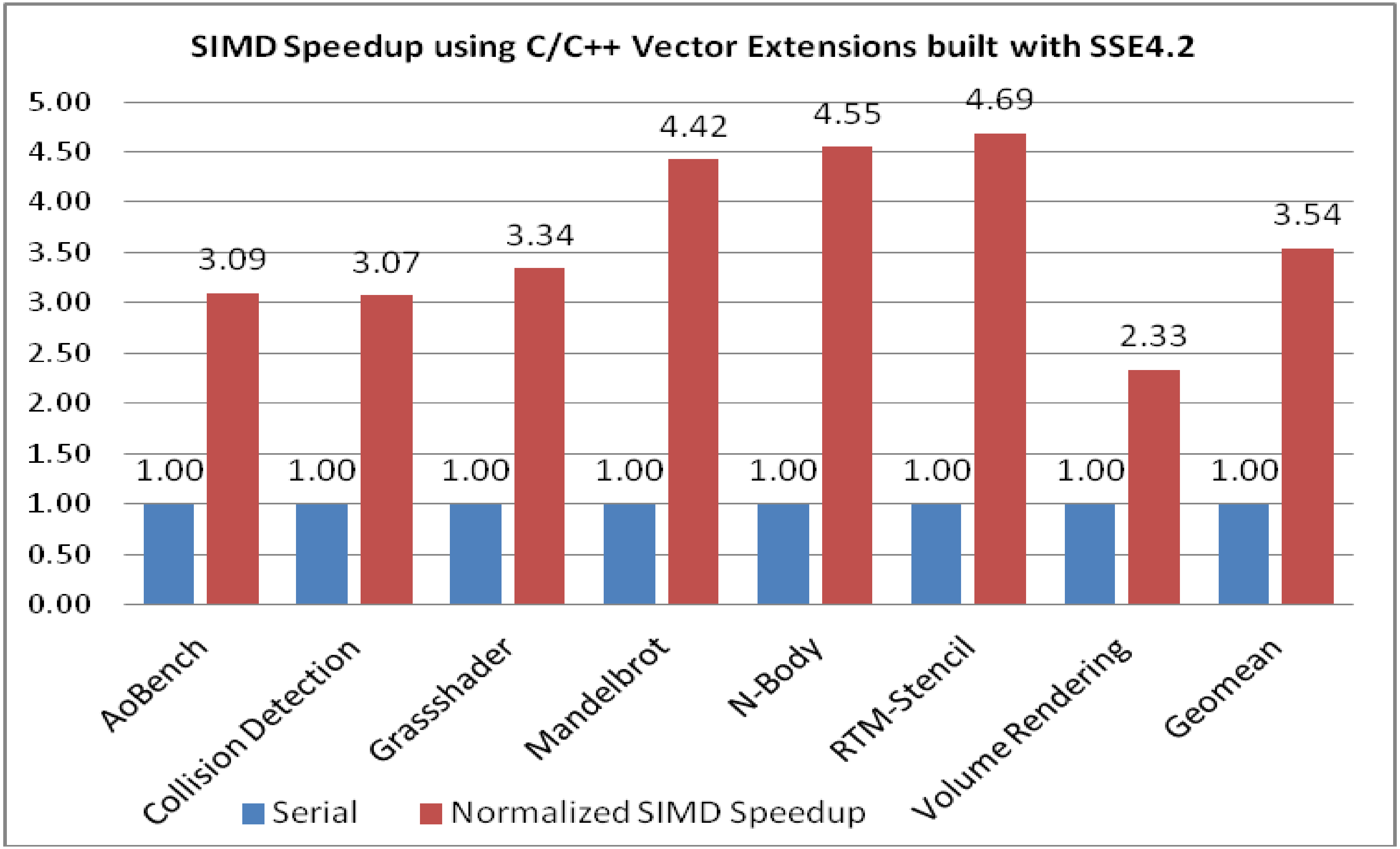
- XMM (128 bit)
  - 16x chars
  - 8X shorts
  - 4x dwords / floats
  - 2X qwords / doubles / float complex
  - Double complex
- YMM (256 bit)
  - 32X chars
  - 16X short
  - 8X dowrd / float
  - 4X qword / double
  - 2X double complex
- AVX- 512
  - 16x chars/shorts (converted to int)
  - 16x dwords/floats
  - 8x qwords/doubles/float complex
  - 4x double complex



3

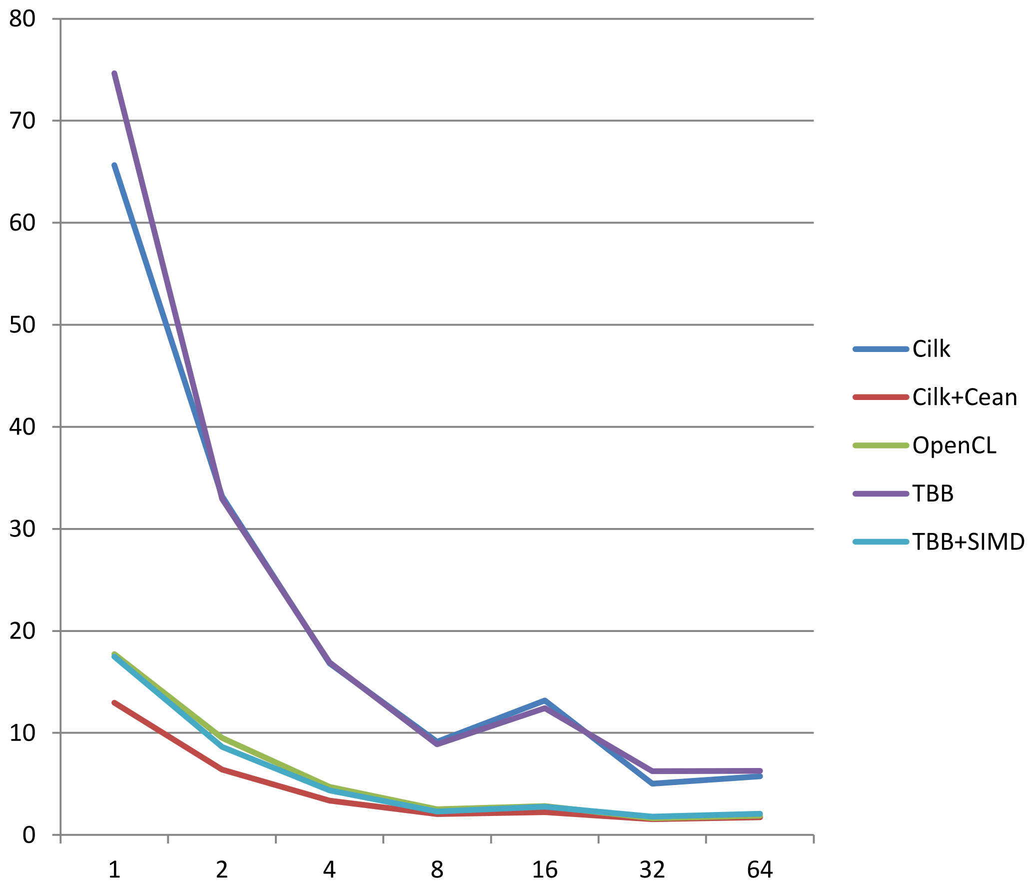
N3734

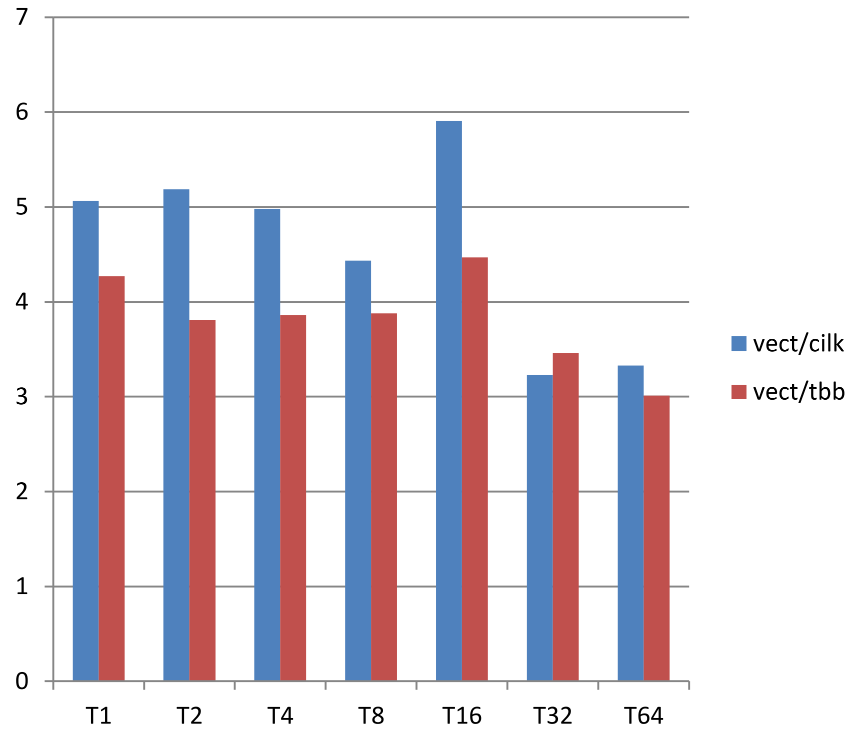
Performance with Vector Parallelism



Measurements by Xinmin Tian for paper in IPDPS, PLC'12

Vectorization performance with 64 threads (a stencil benchmark)





robert.geva@intel.com

Write Vector Code Only Once and Just Recompile for multiple Targets

```
#define ABS(X) \
   ((X) >= 0? (X) : -(X))
int A[1000]; double B[1000];
void foo(int n){
  int i;
  for (i=0; i<n; i++){
    B[i] += ABS(A[i]);
  }
}
```

-O2

```
movq      xmm1, [A+r9+rax*4]
pxor      xmm0, xmm0
pcmpgtd   xmm0, xmm1
pxor      xmm1, xmm0
psubd     xmm1, xmm0
cvtdq2pd  xmm2, xmm1
addpd     xmm2, [B+r9+rax*8]
movaps    [B+r9+rax*8], xmm2
add       rax, 2
cmp       rax, rcx
jb        .B1.4
```

ABS sequence

2 elements

-O2 -QxAVX

```
vpabsd    xmm0, [A+r9+rax*4]
vcvtdq2pd ymm1, xmm0
vaddpd    ymm2, ymm1, [B+r9+rax*8]
vmovupd   [B+r9+rax*8], ymm2
add       rax, 4
cmp       rax, rcx
jb        .B1.4
```

4 elements

-QxSSSE3

```
movq      xmm0, [A+r9+rax*4]
pabsd     xmm1, xmm0
cvtdq2pd  xmm2, xmm1
addpd     xmm2, [B+r9+rax*8]
movaps    [B+r9+rax*8], xmm
add       rax, 2
cmp       rax, rcx
jb        .B1.4
```

ABS instruction

2 elements

Vector Programming

Vector Loops

- Loop iterations execute in "vector order" and use vector instructions

Elemental functions

- Compiled as if part of a vector loop

Array Notations

- Element-wise operations on arrays with vector order semantics

Language extensions to express vector parallelism

Programming vs. Hinting

- Vector programming is a part of parallel programming
- Language syntax provided for “go ahead and generate vector code” model
  - If the results $\neq$ scalar code then it may be a programmers bug, rather than a compiler bug
- Additional constructs include private, reduction, linear, etc

	directive	hint
vector	SIMD	IVDEP
thread	OpenMP	PARALLEL

Not all pragmas are hints

Capabilities in vector loops

Capability	Syntax	Meaning
Vector loop	simd_for (; ;)	Vector order of evaluation
Limit the chunk size	simd_for_chunk(N) (; ;)	Limit the number of iterations that can be grouped together and execute in a chunk
A uniform variable		A single object common for all iterations in a chunk. If iters assign diff values the behavior is undefined
A private variable		Each iteration has a separate object
induction	simd_for (; ; x+=s, p+=4)	A single object across all iteration, and they are allowed to all increment (exception to uniform).
Reduction	TBD, consistent with proposal for reduction in tasking	A single object for all vector lanes, allowed to be modified differently by different iterations, value undefined during the loop, available after the loop.
Turn vector ordering off	simd_off { }	Turn off the relaxed order of evaluaiton within the scope and re-impose C11 order of evaluation.
Elemental Functions	T f (args) simd (qualified args)	Consecutive iterations of the function are chunked and execute together, as if they were in the body of a vector loop

robert.geva@intel.com

Capabilities in Elemental Functions

Capability	Syntax	Meaning
Elemental function	T f (args) simd(args, chunk(N)) { body }	N Consecutive invocations of f are chunked and execute in vector order
A varying parameter	default	Values of the arg within the chunk are unrelated to each other
A uniform parameter	uniform arg	All N values of the arg are the same
A Linear parameter	linear arg:s	N values of args are linear increments by S
Chunk size	chunk(N)	Determines the numbers of consecutive invocations to be grouped together
Multiple versions	T f (args) simd(args, chunk(N1)) simd(args, chunk(N2)) { body }	Multiple versions of f are generated, differ by argument qualifiers and /or chunk size

robert.geva@intel.com

Vector Loops Semantics

- The loops has to be “countable”
- The loop has *logical iterations* numbered 0, 1, … ,N-1
- Order of evaluation:
  - If X is sequenced before Y in the body of the loop, then for each iteration i, X_i is sequenced before Y_i
  - For every X and Y evaluated as part of the vector loop, if X is sequenced before Y and i<j then X_i is sequenced before Y_j
- If the chunk c≥1 is specified then in addition:
  - For every expression X and every iteration i, X_i is sequenced before X_{i+c}
- Note:
  - The above allows order of evaluation that facilitates generation of vector code,
  - it also allows the regular, “scalar” order
  - i.e. vector order of evaluation is not mandated

Different order of evaluation from sequential and from parallel loops

Illustration: Vector Order of Evaluation

```
simd_for (int n = 0; n < N; ++n) {
    a[n] += b[n];
    c[n] += d[n];
}
```

(Remainder loop is left as an exercise for the reader)

```
for (int n = 0; n < N; n+=2) {
    t1 = a[n]; t2 = a[n+1]; // a[n+1] can be written
                            // before c[n] and d[n] are read
    t5 = b[n]; t6 = b[n+1];
    t1 += t5; t2 += t6;
    a[n] = t1; a[n+1] = t2;
    t3 = c[n]; t4 = c[n+1]; // c[n+1] can only be accessed
                            // after a[n]
    t5 = d[n]; t6 = d[n+1];
    t3 += t5; t4 += t6
    c[n] = t3; d[n] = t4;
}
```

Uniform vs. Private variables

```
float m = 3.6f;
float *p = a;
int s = 4;

simd_for (int i = 0; i < N; ++i, p+=s) {
    float tmp = 0.0;
    tmp = *p * m;
    b[i] += tmp;
}
```

- In this example m is uniform: a single object shared between all iterations within a chunk.
- tmp is private: each iteration has a distinct object.
- Different iterations within a chunk cannot assign different values to a uniform variable.

robert.geva@intel.com

Data in Vector Loops

```
float sum = 0.0f;
float *p = a;
int step = 4;
#pragma omp simd

for (int i = 0; i < N; ++i) {
        sum += *p;
        p += step;
}
```

- The two statements with the += operations have different meaning from each other
- The programmer should be able to express those differently
- The compiler has to generate different code
- The variables *i*, *p* and *step* have different “meaning” from each other

Data in Vector Loops

```
float sum = 0.0f;
float *p = a;
int step = 4;
#pragma omp simd reduction(+:sum) \
Linear (p:step)
for (int i = 0; i < N; ++i) {
        sum += *p;
        p += step;
}
```

- The two statements with the += operations have different meaning from each other
- The programmer should be able to express those differently
- The compiler has to generate different code
- The variables *i*, *p* and *step* have different "meaning" from each other

Outer Loop Vectorization

```
simd_for (i=0; i<n; i++) {
    complex<float> c = a[i];
    complex<float> z = c;
    int k = 0;
    while ((k < max_cnt)
          && (abs(z)< limit)) {
          z = z*z + c;
          k++;
    };
    color[i] = k;
}
```

- Each iteration of the(outer) vector loop executes its own version of the (inner) while loop.
- The trip counts of the inner loops are unrelated to each other.
- Each has its own instance of “k”.
- Masking may be required for inactive vector lanes

Outer Loop Vectorization as a motivating example for a uniform qualifier

(*not implemented yet)

```
simd_for (int i=0; i<N; ++i) {
    for (int j = 0; j < M; ++j) {
        a[i][j] = (a[i][j-1] + a[i][j+1])/2;
    }
    b[i] += a[i][N/2];
}
```

- All iterations of the inner loop are the same
- If each iteration has its own instance of “j” then this is not expressed.
- Allow the programmer to express that the inner loop have the same trip count by allowing the declaration of “j” as <u>uniform</u>

In-order Blocks

```
simd_for (int n = 0; n < N; ++n) {
    a[n] += b[n];
    simd_off {
        g1+=a[n];
        g2+=b[n];
    }
}
```

Turn off the vector order of evaluation within the scope of the {}
Enforce scalar order of evaluation
Useful when a portion of the loop is semantically non vectorizeable
For example append noted to a linked list

In-order blocks of code are useful for non-vectorizeable code within loops, where the rest of the loop vectorizeable.

Elemental Functions

- Write a function to describe an operation for one element
- Add __declspec(vector) to get vector code for it

- Then deploy the function across a collection of elements, e.g. arrays
- Each invocation will produce a vector of results instead of a single result

```
float foo(float a, float b, float c, float d) simd()
{
    return a * b + c * d;
}
------------------------------
     vmulps ymm0, ymm0, ymm1
     vmulps ymm2, ymm2, ymm3
     vaddps ymm0, ymm0, ymm2 // vector of results
     ret
```


robert.geva@intel.com

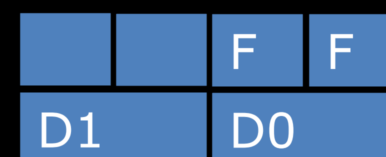
Chunk Size

- How many vectorized copies of the function should execute together per function call?
- As many as you can fit into the hardware vector register
- Constraints: this ratio must be determined consistently yet independently for the function declaration and its callers → cannot rely on the code inside the function, only return type and parameters
- The cases of v_add_f and v_add_d are handled as expected
- In “oops”, most of the time is being spent in single precision, but the compiler cannot automatically use it as the “characteristic type” of the function
- The clause chunk is provided for override
- Another motivation is for correctness
- The use of the chunk clauses changes the linkage of the function

```
float v_add_f(float b, float c) simd()
{
   return b+c;
}
```


```
double v_add_d(double b, double c) simd()
{
         return b+c;
}
```


```
double oops(double e, double f) simd()
{
         return
         sinf(float(e)*sinf(float(f))
}
```



robert.geva@intel.com

Uniform/Linear clauses

- One motivating use case is in address computation
- Can make the difference between vector ld / st (efficient) vs. gather / scatter (less efficient) or multiple scalar loads and merge

```
__declspec(vector)
void foo(float *a, int i);
```

a is a vector of pointers

i is a vector of integers

a[i] becomes gather/scatter

The slow version may defeat the purpose of vector programming altogether

```
__declspec(vector( uniform (a)))
void foo(float *a, int i);
```

a is a pointer

i is a vector of integers

a[i] becomes gather/scatter

```
__declspec(vector( linear(i)))
void foo(float *a, int i);
```

a is a vector of pointers

i is a sequence of integers [i, i+1, i+2…]

a[i] becomes gather/scatter

```
__declspec(vector(uniform(a),linear(i)))
void foo(float *a, int i);
```

a is a pointer

i is a sequence of integers [i, i+1, i+2…]

a[i] is a unit-stride load/store ([v]movups)

BEST PERFORMING OPTION

Multiple versions: Illustration

```
void
vec_add ( float *r, float *op1, float *op2, int i)
    simd (chunk(N))
    simd (uniform (r,op1, op2) , linear (i), chunk(N))
{
    r[i] = op1[i] + op2[i];
}
```

Two vector versions and one scalar

```
simd_for (int i = 0; i<N; ++i) {
    vec_add(a,b,c,i);
}
```

Call matches the version with the uniforms

```
simd_for (int i = 0; i<N; ++i) {
    vec_add(a[x1[[i]],b[x2[[i]],c[x3[[i]],i);
}
```

Call matches the version w/o the uniforms

Invoking Elemental Functions

Construct	Example	Semantics

Sequential for loop	`for (j = 0; j < N; j++) {`
	`    a[j] = my_ef(b[j]);`
	`}`
		Single thread,
		auto vectorization

Vector loop	`simd_for (j = 0; j < N; j++) {`
	`    a[j] = my_ef(b[j]);`
	`}`
		Single thread,
		vectorized, use the vector version if matched

parallel loop	`cilk_for (j = 0; j < N; j++) {`
	`    a[j] = my_ef(b[j]);`
	`}`
		Both vectorization and concurrent execution

Array notation	`a[:] = my_ef(b[:]);`	Vectorization

The rest may not be covered in the Chicago meeting, depending on time.

Array notations for C/C++

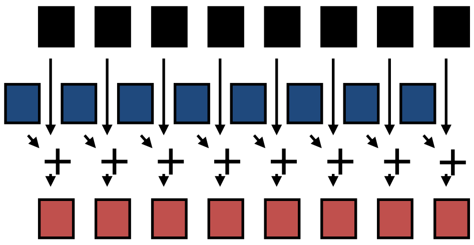
data parallel operations on array sections → vectorization is always semantically correct

```
<array base> [<lower bound>:<length>[:<stride>]]+
```

```
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
```

```
A[:]  = B[:] + C[:]
```

All language standard arithmetic and logical operations.



C /C++ syntax with guaranteed vector implementation

Array Section

```
float a[10];
..
  = a[:];
..
```

a:

0	1	2	3	4	5	6	7	8	9

Array Section

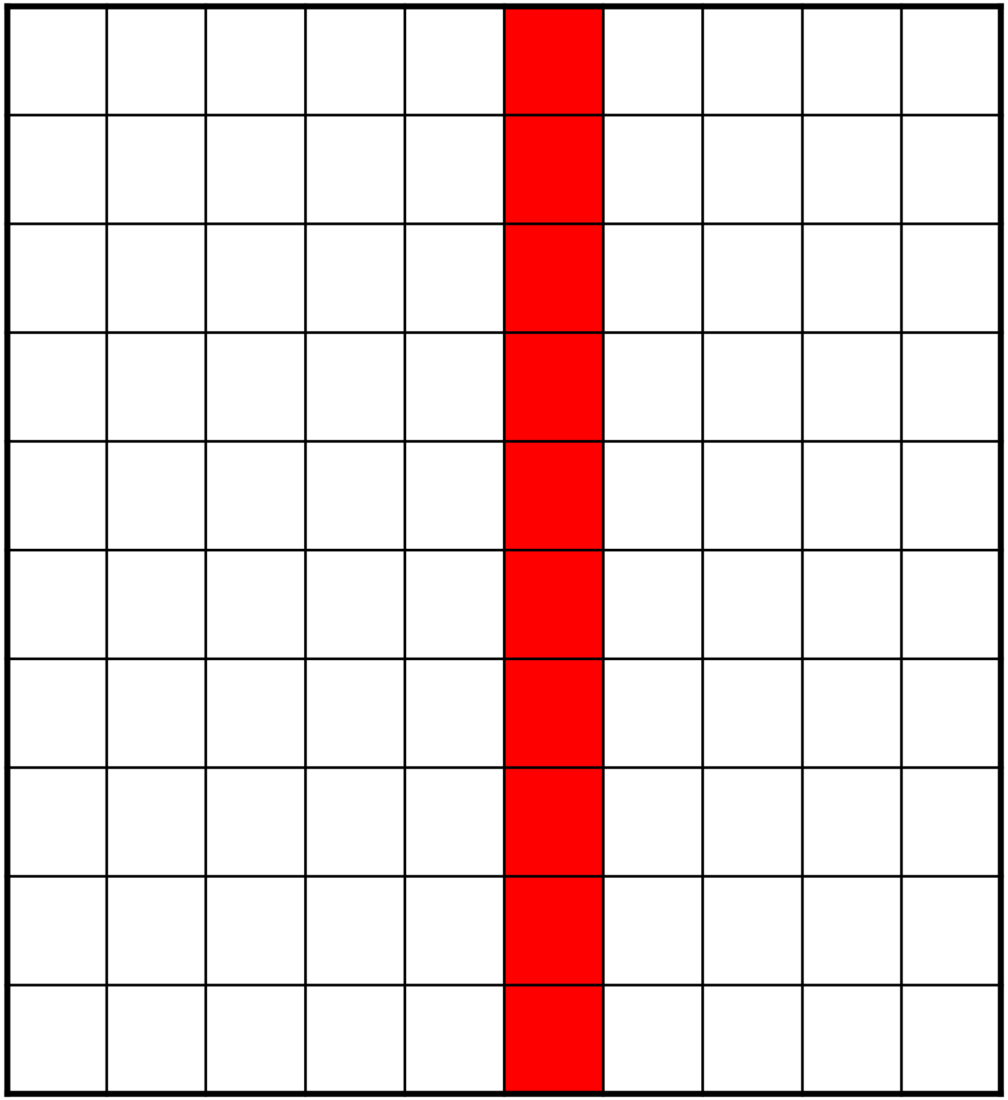
```
float b[10];
..
  = b[2:6];
..
```

b:

Array Section

```
float c[10][10];
..
  = c[:][5];
..
```

c:



robert.geva@intel.com

Array Section

```
float d[10];
..
  = d[0:3:2];
..
```

d:	0	1	2	3	4	5	6	7	8	9

Operator Maps

- Most arithmetic and logic operators for C/C++ basic data types are available for array sections:

```
+, -, *, /, %, <,==,!=,>,|,&,^,&&,||,!,-(unary),
+(unary),++,--, +=, -=, *=, /=, *(p)
```

- An operator is implicitly mapped to all the elements of the array section operands:

```
a[0:s]+b[2:s] => {a[i]+b[i+2], forall
(i=0;i<s;i++)}
```

  - Operations are parallel among all the elements
  - Array operands must have the same *rank* and *extent*
  - Scalar operand is automatically expanded to fill the whole section

```
a[0:s]*c => {a[i]*c, forall (i=0;i<s;i++)}
```


robert.geva@intel.com

Assignment

- Assignment maps to all elements of the LHS array section in parallel:

```
a[:][:] = b[:][2][:] + c;
e[:] = d;
e[:] = b[:][1][:];        // error
a[:][:] = e[:];    // error
```

- LHS of an assignment defines an array context where RHS is evaluated.
  - The rank of the RHS array section must be the same as the LHS
  - The length of each rank must match the corresponding LHS rank
  - Scalar is expanded automatically
- In case of partial overlap between RHS and LHS, results are undefined.
  - Save the temp arrays, deliver higher performance

```
a[1:s] = a[0:s] + 1  // Undefined behavior
```

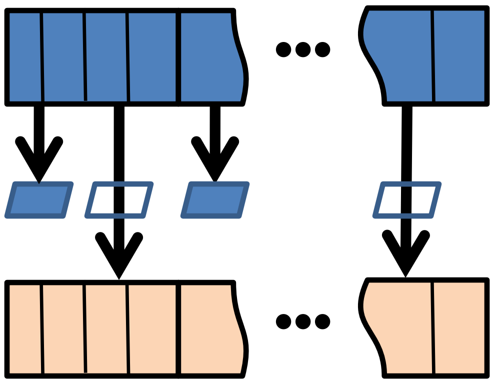

robert.geva@intel.com

Conditional Statements

"If statement" creates a masked vector operation

```
if (a[:] > 0) {
  b[:] = 1;
} else {
  b[:] *= d[:];
}
```

Statements from both "then" and "else" may execute

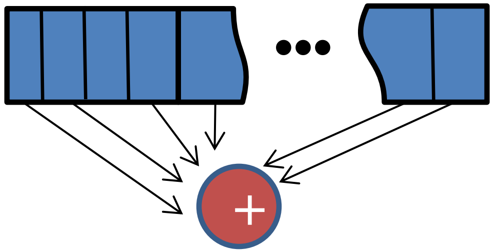


robert.geva@intel.com

Reductions

- Reduction combines array section elements to generate a scalar result

```
int a[] = {1,2,3,4};
sum = __sec_reduce_add(a[:]); // sum
                              // is 10
```



- Nine built-in reduction functions supporting basic C data-types:
  - add, mul, max, max_ind, min, min_ind, all_zero, all_non_zero, any_nonzero

- Supports user-defined reduction function

```
type fn(type in1, type in2); // scalar reduction function
out = __sec_reduce(fn, identity_value, in[x:y:z]);
```

  - Built-in reductions provide best performance

Gather/Scatter

- Take non-consecutive array elements and “gather” them into consecutive locations, or vice-versa.
- The indices of interest are in an *index array*

Gather

```
c[:] = a[b[:]];      // gather elements of a into c,
                     // according to index array b
```

Scatter

```
a[b[:]] = c[:];      // scatter elements of c into a,
                     // according to index array b
```


robert.geva@intel.com

Shift/Rotate

```
b[:] = __sec_shift_right(a[:],  shift_val, fill_val)
b[:] = __sec_shift_left(a[:],   shift_val, fill_val)
b[:] = __sec_rotate_right(a[:], rotate_val)
b[:] = __sec_rotate_left(a[:],  rotate_val)
```

- Shift elements in a[:] to the right/left by *shift_val*
- The leftmost/rightmost element will get *fill_val* assigned
- Rotate will circular-shift elements in a[:] to the right/left by *rotate_val*
- Result is assigned to b[:]
- Argument a[:] is not modified

Shuffle

```
b[:] = __sec_shuffle(a[:], perm)
```

- Permute elements in the array section a[:] and copy the result into b[:].
- The parameter perm is a const array of integer values, which contains the permutation indices to apply to the source.

```
const int perm[] = {3, 2, 1, 0};
…
for (i = 0; i < MAX-4; i+=4) {
   b[i:4] = __sec_shuffle(a[i:4], perm)
}
Resulting in:
…
b[i+0] = a[i + 3];
b[i+1] = a[i + 2];
b[i+2] = a[i + 1];
b[i+3] = a[i + 0];
```

Rank and Shape

- An array section doesn't have a new kind of type
  - the type of an array section is exactly that of the analogous subscript expression.
  - Additionally, an array section has rank and shape.
- A section implicitly iterates over some elements of an array.
  - Rank is the number of levels of loop nesting (i.e. dimensions) in the iteration space.
  - Shape is a (mathematical) vector of lengths. (The rank is the same as the length of the shape vector.)

robert.geva@intel.com

Rank and Shape (continued)

- The rank of an expression is determined statically. In general the shape of a section is determined dynamically.

Expression	Rank	Shape
`a[0]`	0	
`a[0:n]`	1	n
`a[0][i:10]`	1	10
`a[i:n][j:m]`	2	n×m

robert.geva@intel.com

Shapes have to match

- If array size is not known, both lower-bound and length must be specified
- Section ranks and lengths ("shapes") must match.
  - Scalars are OK.

```
a[0:5] = b[0:6]; // No. Size mismatch.
a[0:5][0:4] = b[0:5]; // No. Rank mismatch.
a[0:5] = b[0:5][0:5]; // No. No 2D->1D
a[0:4] = 5; // OK. 4 elements of A filled w/ 5.
a[0:4] = b[i]; // OK. Fill with scalar b[i].
a[10][0:4] = b[1:4]; // OK. Both are 1D sections.
b[i] = a[0:4]; // No. 1D → 0 D
```


robert.geva@intel.com

Array Notation Example

Serial Example

```
float dot_product(unsigned int sz,float A[], float B[])
{
    float dp=0.0f;
    for (int i=0; i<size; i++)
         dp += A[i] * B[i];
    return dp;
}
```

Array Notation Version

```
float dot_product(unsigned int sz,float A[], float B[])
{
    return __sec_reduce_add(A[0:sz] * B[0:sz]);
}
```

Intrinsic reduction

Array Section

Element-wise multiplication

N3734

Vector Programming Summary

- Vector programming is part of parallel programming
- New syntax provided to express vector semantics
- Source code is independent of target architecture
- Currently provided by the Intel compilers, expecting soon in additional compilers
- Standardized as part of OpenMP® 4.0
- Being proposed to the C and C++ committees

robert.geva@intel.com

intel®

Legal Notices

Optimization Notice

Optimization Notice

Intel's compilers 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 SSE2®, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Vector Instructions are Sometimes Smarter (not just wider)

```
#define MAX(x,y) ((x)>(y)?(x):(y))
#define MIN(x,y) ((x)<(y)?(x):(y))
#define SAT2SI16(x) \
  MAX(MIN((x),32767),-32768)
short A[N];

for (i=0; i<n; i++) {
    A[i] = SAT2SI16(A[i]+B[i]);
}
```

Saturating Add

```
movsx     r11d, [rdx+r9*2]
movsx     ebx, [r8+r9*2]
add       r11d, ebx
cmp       r11d, 327
cmovge    r11d, eax
cmp       r11d, -327
cmovl     r11d, ecx
mov       [rdx+r9*2], r11w
inc       r9
cmp       r9, r10
jb        .B1.8
```

11 insts / 1 elem

```
movdqa    xmm0, [rdx+rax*2]
paddsw    xmm0,  [r8+rax*2]
movdqa    [rdx+rax*2], xmm0
add       rax, 8
cmp       rax, r9
jb        .B1.4
```

6 insts / 8 elems

Auto-Vectorization – Limited by Serial Semantics

```
for(i=0;i<*p;i++) {
  a[i] = b[i]*c[i];
  sum = sum + a[i];
}
```

Compiler checks for

- Is "*p" loop invariant?
- Are a, b, and c loop invariant?
- Does a[] overlap with b[], c[], and/or sum?
- Is "+" operator associative? (Does the order of "add"s matter?)
- Vector computation on the target expected to be faster than scalar code?

- Also:
  - How do you vectorize an outer loop
  - How do you allow function calls in vector loop?
  - What if "idiom recognition" fails?

Auto vectorization is limited by the language rules: you can't say what you mean!

Multiple versions

- Multiple declspec(vector) lines are allowed for a single function
- Each will result in another compiled version of the function
- Example: the same function may be called with uniform / non uniform arguments
- Avoiding the second line will deliver correct results but lose performance
- If only the line with uniform is given, then for call sites where the actual arguments are not uniform, the compiler will call the scalar, not vector, version of the function!

```
__declspec(vector)
__declspec(vector(uniform(b,c))
float vmul(float a, float b, float c)
{
   return sqrt(a)*sqrt(b) +
          sqrt(a)*sqrt(c) +
          sqrt(b)*sqrt(c);
}
```