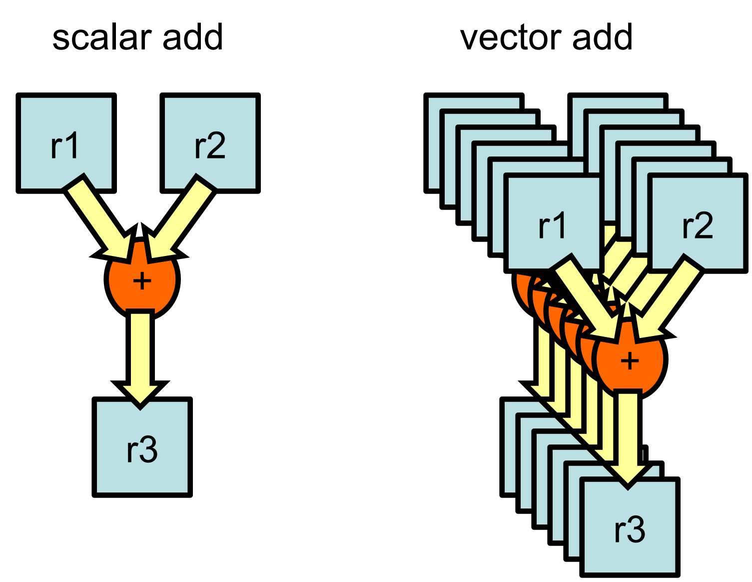
Vector Programming

Nikos Hardavellas

Some slides/material from:
Robert Geva (Intel)

What is a vector instruction?



GPUs vs. SIMD Units (Vectors) in Processors

GTX-680	AVX2 in Haswell
8 SMs	28 cores
4 warp schedulers/SM schedule warps on SPs (32 schedulers on chip)	1 scheduler/core schedules warps on vector unit lanes (28 schedulers on chip)
32 SPs/warp	8 lanes/warp (float)
6 warps/SM	1 warp/core
8*32*6 = 1536 SPs/chip	28*8 = 224 lanes/chip (float)
up to 8*6=48 independent programs at each time	up to 28 independent programs at each time
SMs: lightweight	Vectors: powerful
	Intel Phi: 72 cores w/SIMD (1152 float lanes/Phi card)

Vector Lanes in Intel Processors

Xeon Processor	Year	cores (2S)	SIMD (bits)	Lanes (4B)
X5472	2007	8	SSSE3 (128)	32
X5570	2009	8	SSE4.2 (128)	32
X5680	2010	12	SSE4.2 (128)	48
E52690	2012	16	AVX (256)	128
E52697 v2	2013	24	AVX (256)	192
Haswell	2014	28	AVX2 (256)	224
Knights Landing	2016	72	AVX512 (512)	1152

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]);
}
```

```
movsx    r11d, [rdx+r9*2]
movsx    ebx, [r8+r9*2]
add      r11d, ebx
cmp      r11d, 32767
cmovge   r11d, eax
cmp      r11d, -32768
cmovl    r11d, ecx
mov      [rdx+r9*2], r11w
inc      r9
cmp      r9, r10
jb       .B1.8
```

11 insts / 1 elem
88 insts / 8 elems

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

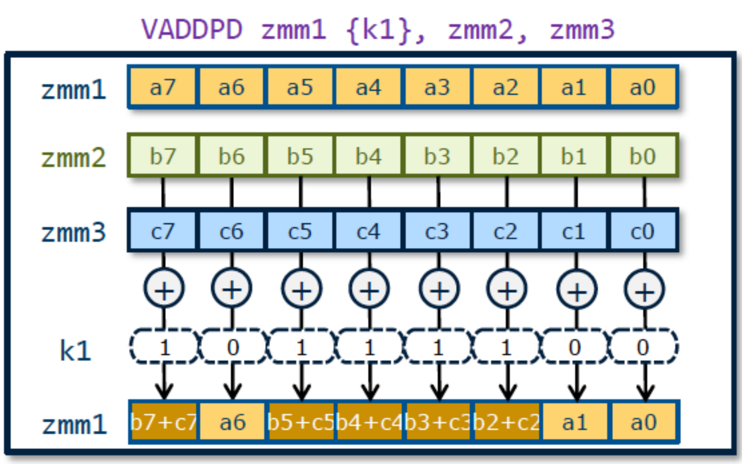
6 insts / 8 elems

Saturating Add

Performance gain of vectorized code:

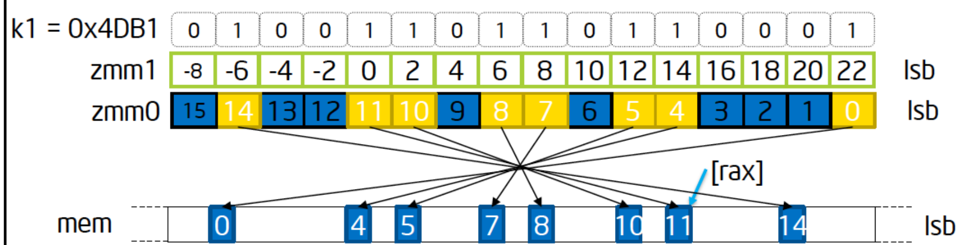
- L1 cache: 16x
- L2 cache: 8x
- Main mem: nada

Example: AVX512 vector add with mask



Example: AVX512 scatter store

```
VPSCATTERDD zmm0, ([rax], zmm1, 4) {k1}
```



How Can Someone Vectorize Code?

Choice 1:

use a compiler switch for auto-vectorization

(and hope it vectorizes)

Compiler-Directed Auto-Vectorization

```
#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
```

-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
```

-QxSSSE3

```
movq      xmm0, [A+r9+rax*4]
pabsd     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
```

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 the “+” operator associative?
- 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!

How Can Someone Vectorize Code?

Choice 2:

give your compiler hints

(and hope it vectorizes)

C99 Restrict Keyword

- For the lifetime of the pointer, only it or a value directly derived from it (such as pointer + 1) will be used to access the object to which it points.
- Limits memory aliasing, enables optimizations

```
void v_add (float *restrict c,
            float *restrict a,
            float *restrict b)
{
     for (int i=0; i<= MAX; i++)
          c[i]=a[i]+b[i];
}
```

IVDEP (ignore assumed vector dependencies)

```
void v_add (float *c, float *a, float *b)
{
#pragma ivdep
     for (int i=0; i<= MAX; i++)
          c[i]=a[i]+b[i];
}
```

How Can Someone Vectorize Code?

Choice 3:

code explicitly for vectors
(mandatory vectorization)

Programming vs. Hinting

- Vector programming is a part of parallel programming
- Language syntax provided for “go ahead and generate vector code” model
  - Vectorization is semantic at the source code level
  - If the results ≠ scalar code then it may be a programmers bug, rather than a compiler bug
- Additional constructs: private, reduction, linear, …

	directive	hint
vector	SIMD #pragma simd	IVDEP
thread	OpenMP #pragma omp	PARALLEL

Vector Programming

Vector Loops	• Iterations execute in “vector order” and use vector instrs.
Simd-enabled functions	• Compiled as if part of a vector loop
Array Notations	• Element-wise operations on arrays with vector semantics
Intel initial syntax	• #pragma simd • As of 2010
OpenMP standards	• #pragma omp simd • Part of OpenMP 4.0
Keyword proposal for C/C++	• for _Simd (; ;) { body } • Supported by version 15.0

Language Based Vectorization: Vector Loops

```
#pragma simd reduction(+:sum)

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

- The programmer write vector code
- With vector semantics at the source level
- The compiler generates vector code
- The programmer needs to pay attention
  - Correctness
  - Efficient vector code

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
- 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

```
for (i=0; i < N; ++i) {
        X;
        Y;
}
```

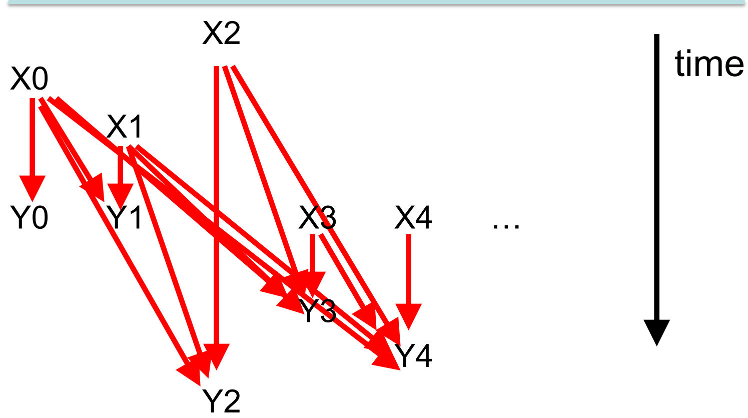


Illustration: Vector Order of Evaluation

```
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)

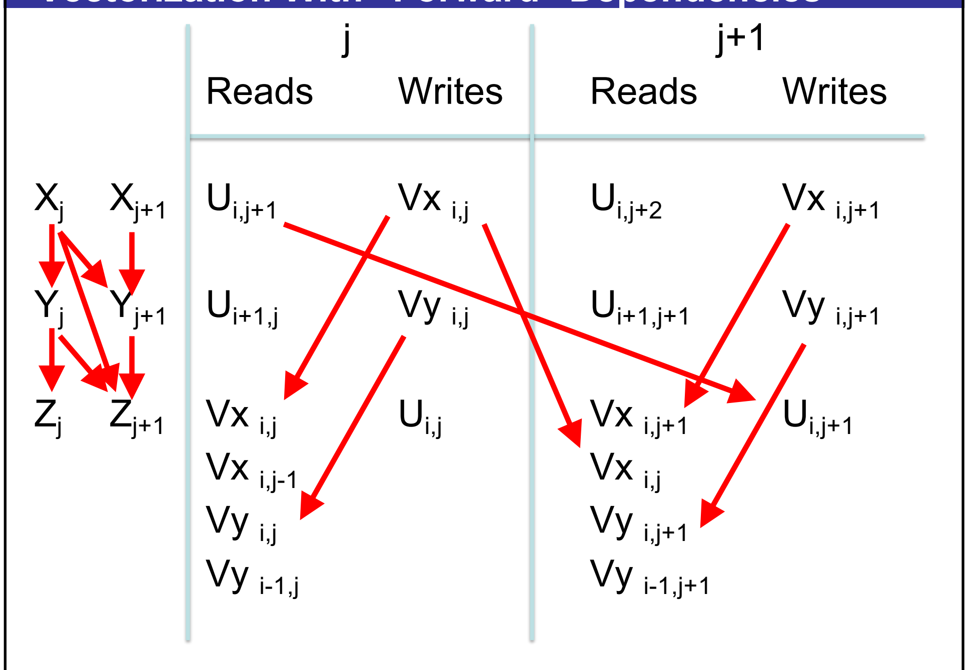
```
for (int n = 0; n < N; n+=2) {
    t1 = a[n]; t2 = a[n+1]; // a[n+1] can be written
                            // before c[n] and d[n] are read
    t5 = b[n]; t6 = b[n+1];
    t1 += t5; t2 += t6;
    a[n] = t1; a[n+1] = t2;
    t3 = c[n]; t4 = c[n+1]; // c[n+1] can only be accessed
            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

Vectorization With “Forward” Dependencies

```
void sweep (int i0, int i1, int j0, int j1 ) {
  for( int i=i0; i<i1; ++i )
#pragma simd
    for( int j=j0; j<j1; ++j ) {
        float u = U[i][j];
        Vx[i][j] += (A[i][j+1]+A[i][j])*(U[i][j+1]-u); // X
        Vy[i][j] += (A[i+1][j]+A[i][j])*(U[i+1][j]-u); // Y
        U [i][j] = u + B[i][j]*((Vx[i][j]-Vx[i][j-1])
                     + (Vy[i][j]-Vy[i-1][j]));         // Z
    }
}
```

- Seismic duck: modeling wave equation
- Exploits the ability to vectorize with “forward” dependencies
- X reads U[i][j+1] and Z writes U[i][j]
- Y reads U[i+1][j] and Z writes U[i][j]
- X writes Vx[i][j] and Z reads Vx[i][j] and Vx[i][j-1]
- Y writes Vy[i][j] and Z reads Vy[i][j] and Vy[i-1][j]

Vectorization With “Forward” Dependencies



Data in Vector Loops

```
float sum = 0.0f;
float *p = a;
int step = 4;

#pragma 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

SIMD-Enabled (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

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

Uniform/Linear Clauses

- uniform: broadcast same value to iterations
- linear: i, i+1, i+2, …
- Most useful in the address computation
- Can make the difference between vector ld / st (efficient) vs. gather / scatter (less efficient)

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

a is a vector of pointers
i is a vector of integers
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
BEST 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];
}
```

OK to execute N iterations in parallel

Two vector versions and one scalar

similar syntax: __declspec() or simd()

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

Call matches the scalar version

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

Call matches the version with the uniforms

```
for _Simd (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

Vectorization With OpenMP Syntax

```
#pragma omp declare simd
int binsearch(int key) {
   int lo = 0; int hi = N; int found = 0; int ans = -1;
   while ((!found) && (lo <= hi)) {
      int mid = lo + ((hi - lo) >> 1);
      int t = sortedarr[mid];
      if (key == t) {
          ans = mid; break;
      } else if ( key > t) {
          lo = mid + 1;
      } else {
          hi = mid - 1;
      }
   }
   return ans;
}
```

```
#pragma omp simd
for (int i=0; i<M; i++) {
   ans[i] = binsearch(keys[i]);
};
```

The Recursive Version

```
#pragma omp declare simd
int binsearch(int key, int lo, int hi) {
    int ans;
    if ( lo > hi) {
        ans = -1;
    } else {
       int mid = lo + ((hi - lo) >> 1);
       int t = sortedarr[mid];
       if (key == t) {
           ans = mid;
       } else if ( key > t) {
           ans = binsearch(key, mid + 1, hi);
       } else {
           ans = binsearch(key, lo, mid - 1);
       }
   };
   return ans;
}
```

```
#pragma omp simd
for (int i=0; i<M; i++) {
   ans[i] = binsearch(keys[i], 0, N-1);
};
```

Example – Search Key(s) In Many String(s)

```
#pragma omp declare simd
bool is_equal(char *data, char *key) {
    int ret_val = 1; char y = *key;
    for (; y != '\0'; data++, key++, y = *key) {
       if (*data != y) {  // if mismatch ever, return 0
            ret_val = 0; break;
        };
    };
    return ret_val;
}
```

```
#pragma omp declare simd
int search_substring(char *data_string, char *key) {
   int ret_val = -1;
   for(int i=0; *data_string != '\0'; i++, data_string++) {
       if (is_equal(data_string, key)) {// match at position i?
           ret_val = i; break;
       };
   };
   return ret_val;
}
```

```
#pragma omp simd
for (int i=0; i<NO_STRINGS; i++) {
    found[i] = search_substring(str_array, keys[i]);
};
```

Outer Loop Vectorization

```
for _Simd (i=0; i<n; i++) {
    complex<float> c = a[i];
    complex<float> z = c;
    int j = 0;
    while ((j < 255)  && (abs(z)< limit)) {
          z = z*z + c;
          j++;
    };
    color[i] = j;
}
```

Each vector lane executes its own version of the inner loop. To implement this, the compiler has to vectorize across the inner loop

Vectorize Outer Loop With Func. Calls – LIBOR example

```
#pragma omp declare simd
static void path_calc_b1(REAL *z, REAL *L, REAL *L2, const
REAL* lambda)
{
  int   i, n;
  REAL sqez, lam, con1, v, vrat;
  memcpy(L2, L, NN*sizeof(REAL));
  for(n = 0; n < NMAT; n++) {
    sqez = SQRT_DELTA * z[n];
    v =REAL(0);
    for (i=n+1; i<NN; i++) {
      lam  = lambda[i-n-1];
      con1 = DELTA * lam;
      v   += con1 * L[i] / (REAL(1) + DELTA * L[i]);
      vrat = std::exp(con1 * v + lam * (sqez - REAL(0.5) *
con1));
      L[i] = L[i] * vrat;
      L2[i+(n+1)*NN] = L[i];
    }
  }
}
```

```
#pragma omp simd reduction(+: sumv) reduction(+: sumlb)
  for (path=0; path<numPaths; path++) {
    path_calc_b1(ptrZ, L, L2, lambda);
    path_calc_b2(L_b, L2, lambda);
  }
```

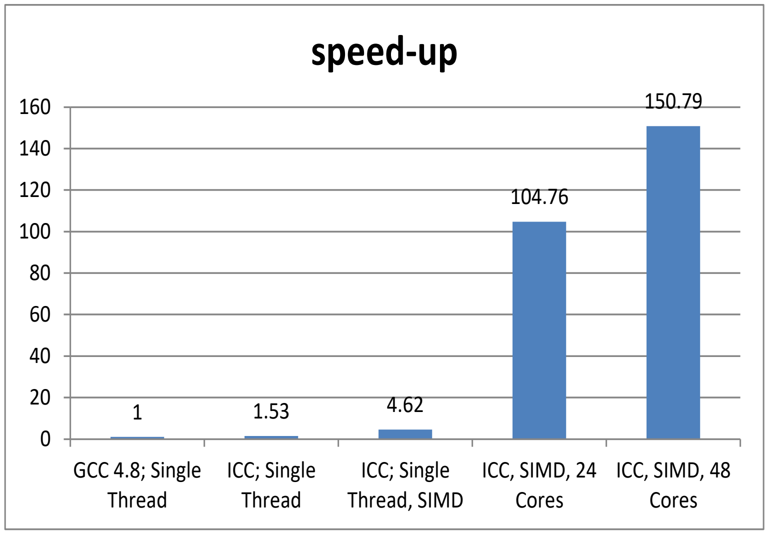
In-order Blocks

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

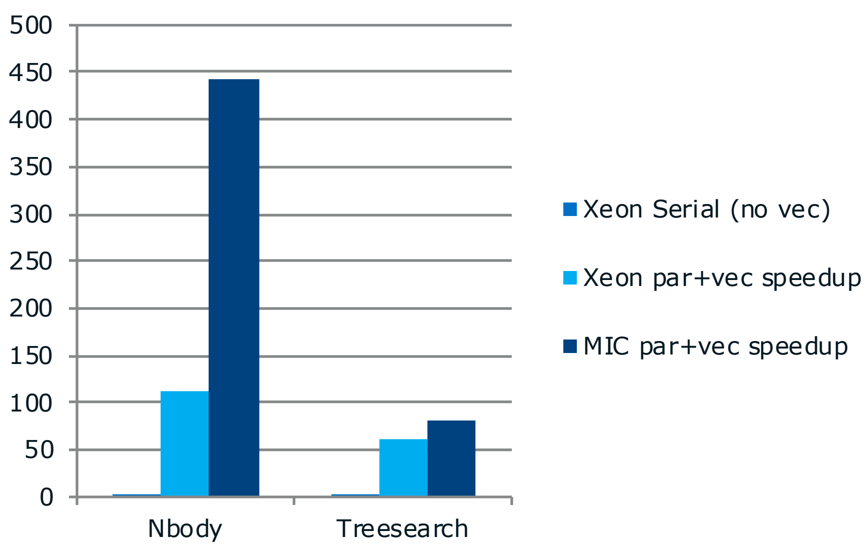
Turn off the vector order of evaluation within the scope of the {}
Enforce scalar order of evaluation
Useful when a portion of the loop is semantically non vectorizeable
For example append noted to a linked list

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

Case Study: Swaption Pricer



Xeon/MIC Performance Comparison (Speedup)



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 several compilers
  - Intel icc, LLVM
- Standardized as part of OpenMP 4.0
- Extensions proposed to the C and C++ committees
- Advanced examples include
  - Vectorization of outer loops
  - Vectorization of recursive functions (fib, search)