Explicit Vectorisation

Stephen Blair-Chappell

Intel Compiler Labs

This training relies on you owning a copy of the following…

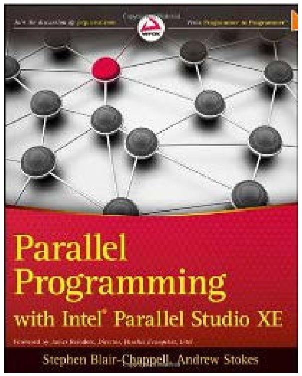
Parallel Programming with Parallel Studio XE

Stephen Blair-Chappell & Andrew Stokes

Wiley ISBN: *9780470891650*

Part I: Introduction

1: Parallelism Today
2: An Overview of Parallel Studio XE
3: Parallel Studio XE for the Impatient



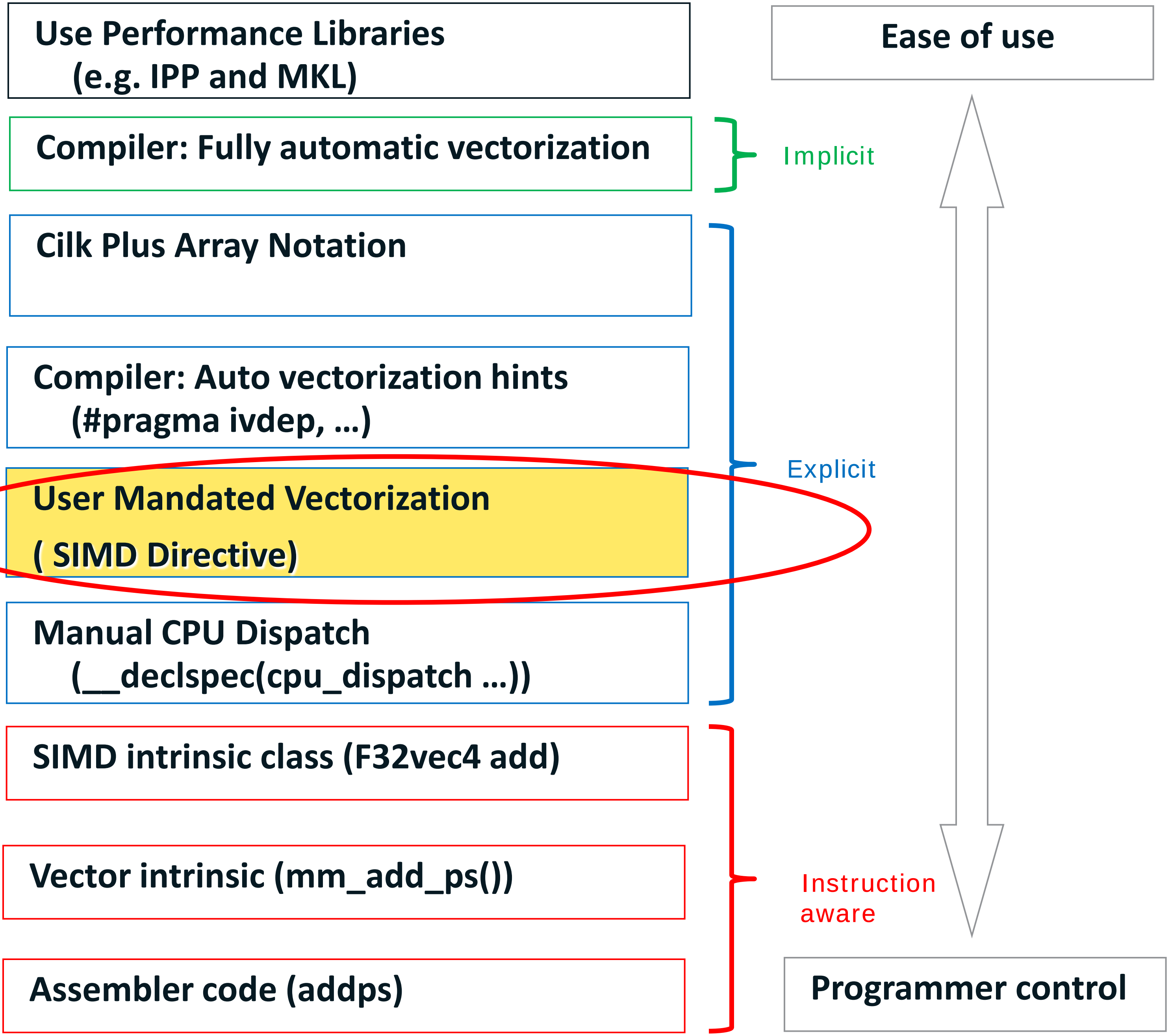
Part II: Using Parallel Studio XE

4: Producing Optimized Code
5: Writing Secure Code
6: Where to Parallelize
7: Implementing Parallelism
8: Checking for Errors
9: Tuning Parallelism
10: Advisor-Driven Design
11: Debugging Parallel Applications
12:Event-Based Analysis with VTune Amplifier XE

Part III :Case Studies

13: The World's First Sudoku 'Thirty-Niner'
14: Nine Tips to Parallel Heaven
15: Parallel Track-Fitting in the CERN Collider
16: Parallelizing Legacy Code

Other Ways of Inserting Vectorised Code



- Use Performance Libraries (e.g. IPP and MKL)
- Compiler: Fully automatic vectorization
- Cilk Plus Array Notation
- Compiler: Auto vectorization hints (#pragma ivdep, …)
- User Mandated Vectorization (SIMD Directive)
- Manual CPU Dispatch (__declspec(cpu_dispatch …))
- SIMD intrinsic class (F32vec4 add)
- Vector intrinsic (mm_add_ps())
- Assembler code (addps)

SIMD Pragma

Language Based Vectorization

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

This loop implies:

- “*p” is loop invariant
- a[] is not aliased with b[], c[], and sum
- sum is not aliased with b[] and c[]
- Generate a private copy of sum for each iteration
- “+” operation on sum is associative (Compiler can reorder the “add”s on sum)
- Vector code to be generated even if it could be slower than scalar code

Optimization Notice

SIMD Pragma: Definition

Top-level

- C/C++: #pragma simd
- Fortran: !DIR$ SIMD

	directive	hint
vector	SIMD	IVDEP
thread	OpenMP	PARALLEL

Attached clauses to describe semantics

- `vectorlength (VL)`
- `private / firstprivate / lastprivate (var1[,var2, …])`
- `reduction (oper1:var1[, …][, oper2:var2[, …]])`
- `linear (var1[:step1][, var2[:step2], …])`

OpenMP*-like pragma for vector programming

A keyword base syntax also being added

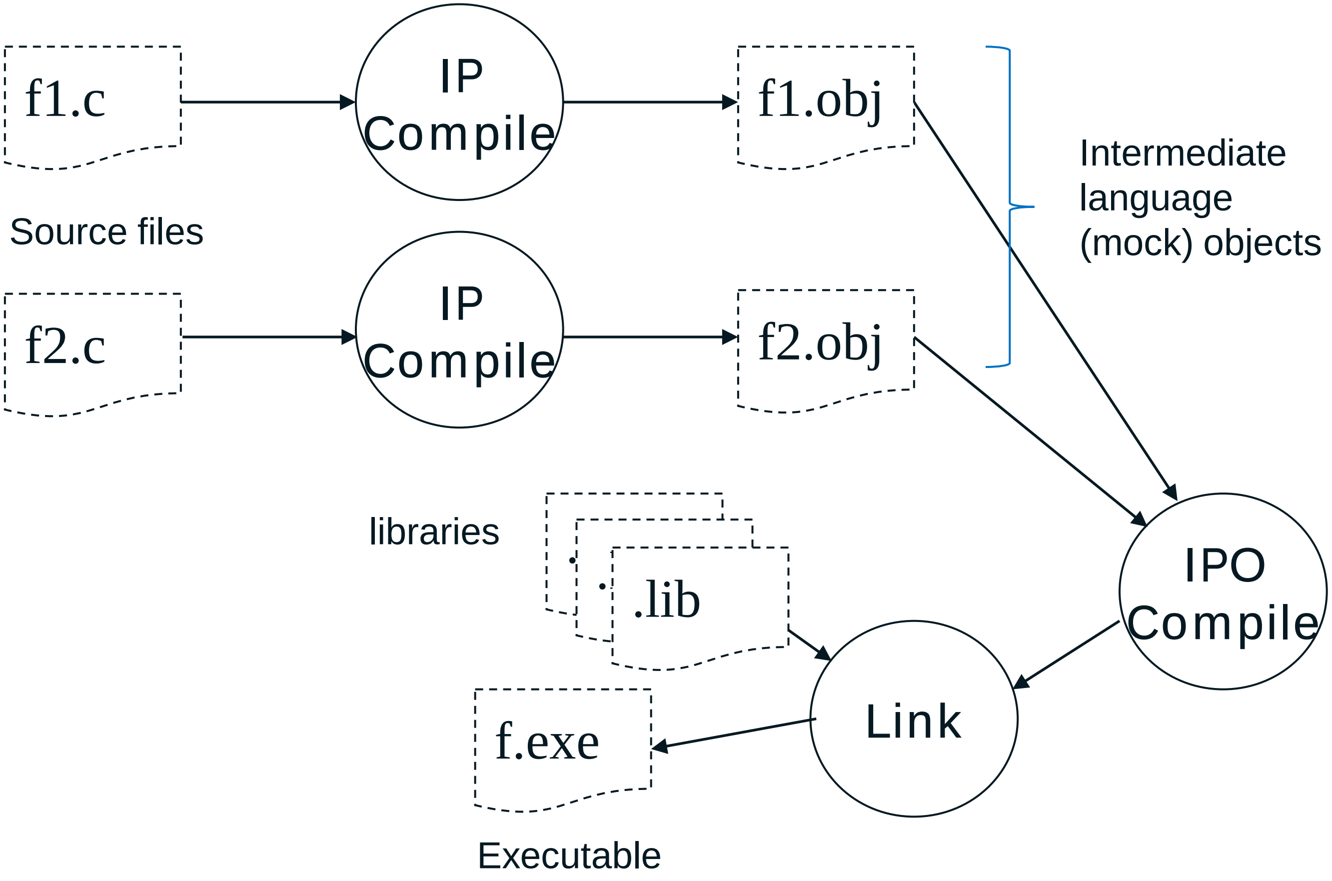
- Not everyone wants to program with pragmas

Step 4

Using Inter Procedural Optimisation

-and its effect on Vectorisation

Interprocedural Optimisation



Step 4

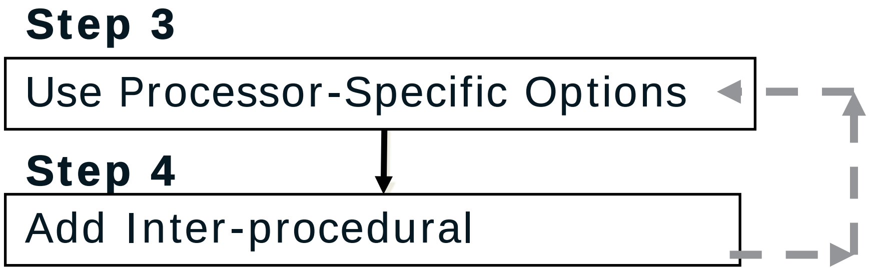
What you should know about IPO

- 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
- *In-lining of functions is 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
- IPO works for libraries too
  - Not trivial topic – see documentation

Step 4

Optimization Notice

Impact of IPO on Auto-Vectorisation



Step 3

Use Processor-Specific Options

Step 4

Add Inter-procedural

- IPO improves auto-vectorization results of the sample application
- IPO brings some new ‘tricky-to-find’ auto-vectorization opportunities.

Step 4

Results of IPO

PLATFORM	O2	IPO	QXHOST	SPEEDUP O2 TO IPO	SPEEDUP O2 TO QXHOST
Core 2 Laptop	0.474	0.272	0.266	1.74	1.78
SNB	0.293	0.181	0.171	1.62	1.71
SNB Turbo	0.211	0.132	0.124	1.60	1.70
Xeon workstation	0.239	0.211	0.209	1.13	1.14

Step 4

Optimization Notice

Consequence of ipo – multiple vectorisation messages about the same line

```
chapter4.c(51): (col. 11) remark: LOOP WAS VECTORIZED.
. . .
chapter4.c(51): (col. 11) remark: loop was not vectorized:
not inner loop.
chapter4.c(51): (col. 11) remark: loop was not vectorized:
not inner loop.
chapter4.c(51): (col. 11) remark: loop was not vectorized:
existence of vector dependence.
```

To see code rather than call sites use

-debug inline-debug-info (Linux)

/debug:inline-debug-info (Windows)

Warning: When using this option always explicitly add -O1, -O2 or –O3, or compiler will assume –O0

Step 4

Modified code results in more improvements

series.c

```
double Series2(int j)
{
  int k;
  double sumy = 0.0;
  for( k=j; k>0; k--)
  {
    // sumy++;
    sumy = AddY(sumy, k);
  }
   return sumy;
}
```

This modification results in a 20% boost of performance

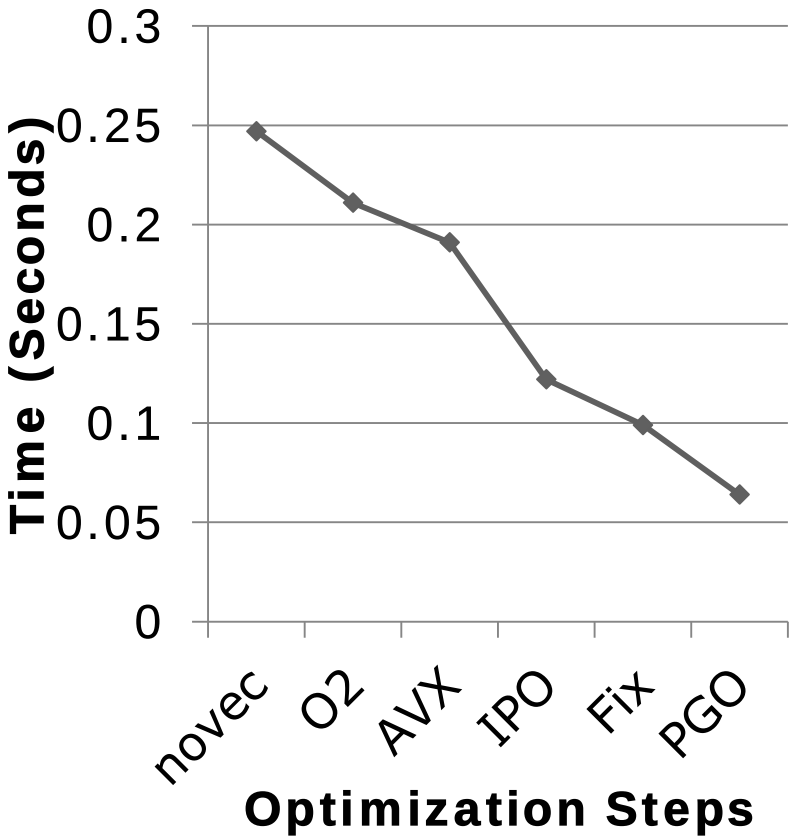
addy.c

```
double AddY( double sumy, int k )
{
// sumy--;
sumy = sumy + (double)k;
  return sumy;
}
```

Step 4

Optimization Notice

Results so far …



Lower is better

Speedup
0.211 / 0.064 = 3.2

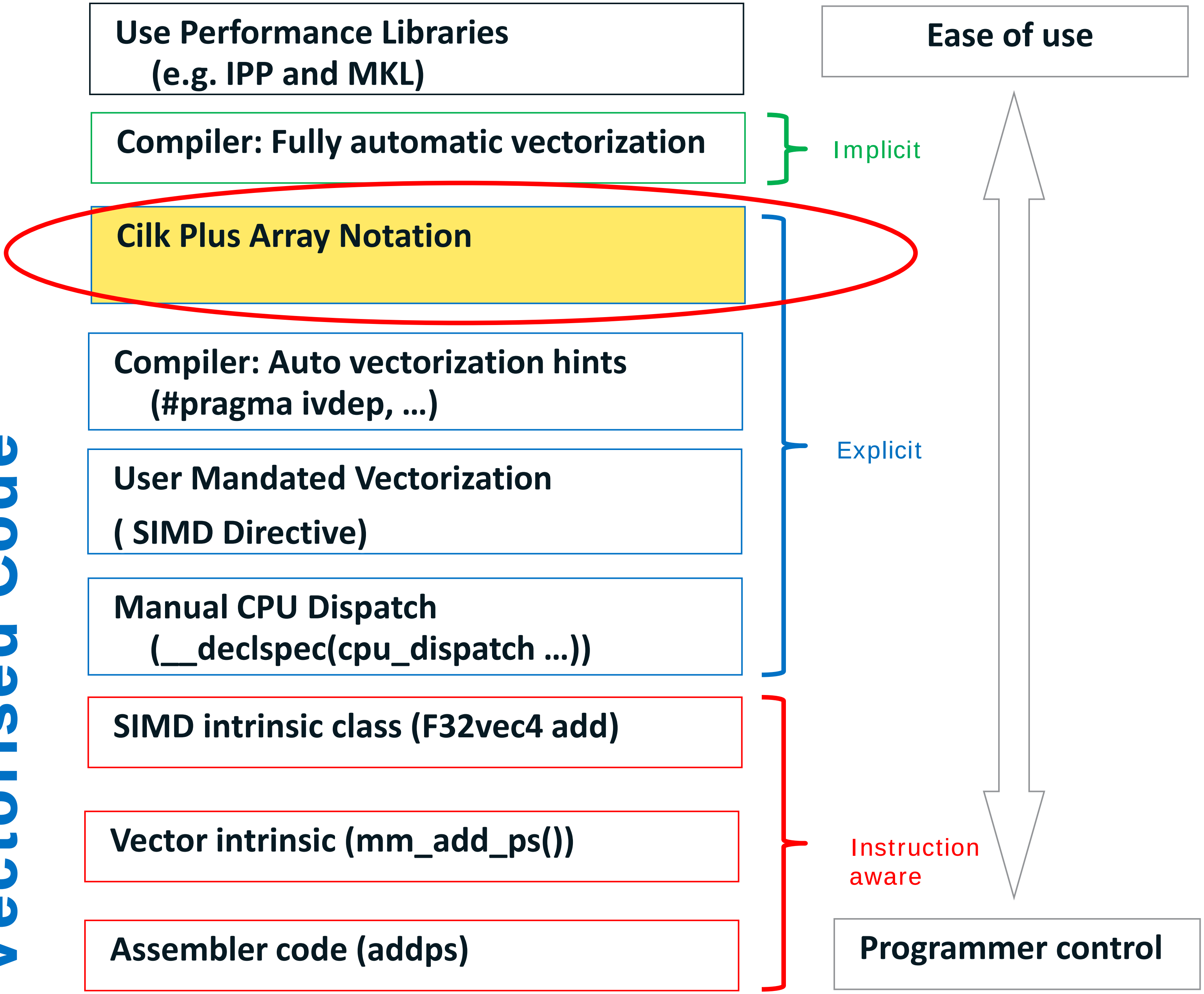
Step 5

Optimization Notice

Additional Info

More on Auto-vectorisation

Other Ways of Inserting Vectorised Code



Use Performance Libraries
(e.g. IPP and MKL)

Compiler: Fully automatic vectorization

Cilk Plus Array Notation

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

User Mandated Vectorization
(SIMD Directive)

Manual CPU Dispatch
(__declspec(cpu_dispatch …))

SIMD intrinsic class (F32vec4 add)

Vector intrinsic (mm_add_ps())

Assembler code (addps)

Optimization Notice

Cilk Plus Array Notation

- An extension to C language to all manipulation of arrays

Array[lower bound : length : stride].

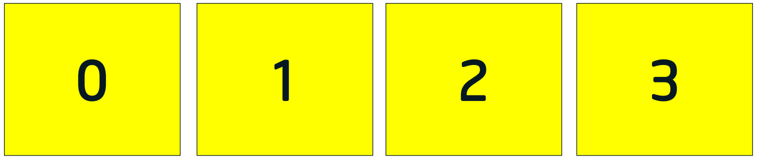
- Main advantages are
  - Easier ('allegedly') to manipulate of arrays
  - Compile will vectorize the code
    - Build at -O1 or higher
    - Default generates SSE2, but can be influenced by /arch, /Qx, or Qax.
- Not yet in any standard, but Intel working hard at this

The Section Operator (:)

Array[lower bound : length : stride].

int A[]4

A[:] // All of array A



The Section Operator (:)

Array[lower bound : length : stride].

int B[13]

B[4: 7] // Elements 4 to 10

0	1	2	3	4	5	6	7	8	9	10	11	12

The Section Operator (:)

Array[lower bound : length : stride].

`int C[3][9]`

C[:][3] // Column 3

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

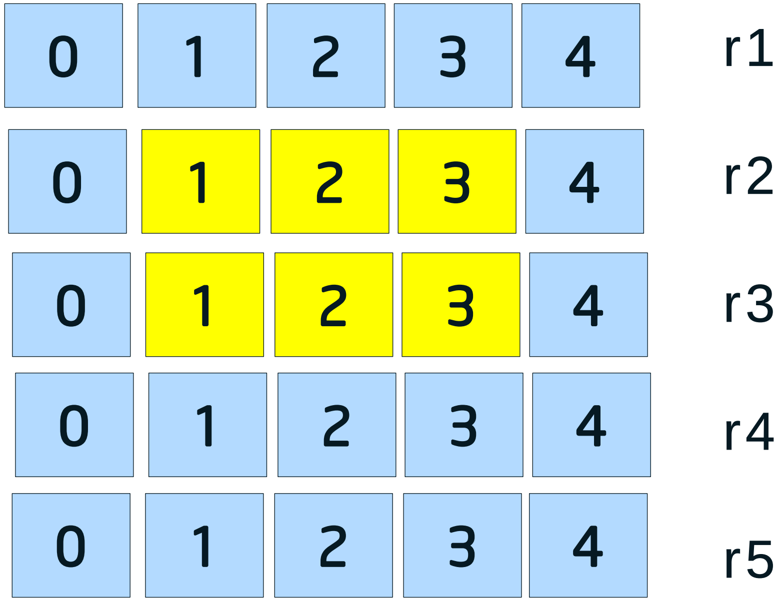
The Section Operator (:)

Array[lower bound : length : stride].

`int C[5][5]`

`C[2:2][ 1:3]`

`// block`



0	1	2	3	4	r1
0	1	2	3	4	r2
0	1	2	3	4	r3
0	1	2	3	4	r4
0	1	2	3	4	r5

The Section Operator (:)

Array[lower bound : length : stride].

D[0: 3: 2]

0	1	2	3	4	5	6	7	8	9	10	11	12

C/C++ Operators

- Most operators supported
- Each operation mapped onto each element of array

```
z[:] = x[:] * y[:]                          // element-wise multiplication

c[3:2][3:2] =
     a[3:2][3:2] + b[5:2][5:2]              // 2x2 matrix addition
```

Equivalent to →

```
c[3][3] = a[3][3] + b[5][5];
c[3][4] = a[3][4] + b[5][6];
c[4][3] = a[4][3] + b[6][5];
c[4][4] = a[4][4] + b[6][6];
```

Array[lower bound : length : stride].

Gather & Scatter

When an array section occurs directly under a subscript expression, it designates a set of elements indexed by the values of the array section.

```
a[b[0:s]] = c[:]    => for(i=0;i<s;i++)
                            a[b[i]]=c[i];

c[0:s] = a[b[:]]    => for(i=0;i<s;i++)
                            c[i]=a[b[i]];
```

Compiler generates scatter and gather instructions on supported hardware for irregular vector access.

Shift & Rotate

These functions shift or rotate all array elements in a given rank-one section by a given amount.

- Shift elements in a[:] to the right (*shift_val*>0) or to the left (*shift_val*<0) by *shift_val* elements. *fill_value* will used to fill in the leftmost/rightmost elements.

```
b[:] = __sec_shift(a[:], shift_val, fill_value);
```

- Circular-shift all elements in a[:] to the right (*shift_val*>0) or to the left (*shift_val*<0) by *shift_val* elements.

```
b[:] = __sec_rotate(a[:], shift_val);
```

Shuffle

Returns a permutation of the given array section.

```
int a[10],b[10],c[4],d[4];
const int perm[10] = {9,8,7,6,5,4,3,2,1,0};
const int perm2[4] = {2,2,0,0};

foo() {
  a[:] = __sec_implicit_index(0)*2;  // a is {0,2,4,6,8,10,12,14,16,18,20}
  b[:] = a[:][perm[:]];              // b is {20,18,16,14,12,10,8,6,4,2,0}
  c[0:4] = a[perm[6:4]];             // c is {6,4,2,0}
  b[0:4] = a[perm2[:]];              // b is {4,4,0,0}
}
```

Reducers

- Accumulate values in array

_sec_reduce_add — Adds values _sec_reduce_mul — Multiplies values
_sec_reduce_all_zero — Tests that all elements are zero
_sec_reduce_all_nonzero — Tests that all elements are nonzero
_sec_reduce_any_nonzero — Tests that any element is nonzero
_sec_reduce_max — Determines the maximum value
_sec_reduce_min — Determines the minimum value
_sec_reduce_max_ind — Determines index of element with max value
_sec_reduce_min_ind — Determines index of element with min value

```
// add all elements using a reducer
int sum = _sec_reduce_add( c[:])

// add all elements using a loop
int sum = 0;
for( int i = 0; i < sizeof(c)/sizeof(c[0]); i ++)
  sum + = c[ i]);
```

Optimization Notice

Function Maps

A scalar function call be mapped to the elements of array section parameters:

```
a[:] = sin(b[:]);
a[:] = pow(b[:], c);      // b[:]**c
a[:] = pow(c, b[:]);      // c**b[:]
a[:] = foo(b[:]);   // user defined function
a[:] = bar(b[:], c[:][:]); //error, different ranks
```

- Functions are executed in parallel.
- The compiler generates calls to vectorized library functions.

Elemental Functions

User defined ‘per element’ functions

Steps:

1. Write ‘normal scalar operation’
   ```
   int multwo( int i){ return i * 2;}
   ```
2. Decorate function with _declspec(vector).
   ```
   int _declspec( vector) multwo( int i){ return i * 2;}
   ```
3. Call Function with Vector Arguments
   ```
   int main()
   {
        int A[ 100]; A[:] = 1;
        for (int i = 0 ; i < 100; i + +)
        multwo( A[ i]);
   }
   ```

Why doesn’t this code build?

```
#define N 2
void MatrixMul( double a[ N][ N], double
b[ N][ N], double c[ N][ N])
{
  int i, j;
  for (i = 0; i < N; i + +)
  {
    for (j = 0; j < N; j + +)
    {
      c[ i][ j] + = a[ i][:] * b[:][ j];
    }
  }
}
```

```
mm.c(9): error: rank mismatch in array section expression
      c[ i][ j] += a[ i][:] * b[:][ j];
                ^
```

Every expression has a *rank*, determined as follows.
The rank of an expression with no nested sub-expression is zero. (This rule applies to identifiers and constants.)

Unless otherwise specified, the rank of an expression with one sub-expression operand is the rank of its operand. (This rule applies to parenthesized expressions, most postfix expressions, most unary expressions, and cast expressions.)

Unless otherwise specified, in an expression with more than one sub-expression operand, the rank is the common rank of its operands. The *common rank* of two expressions is

- if the rank of either expression is zero, the common rank is the rank of the other expression;
- otherwise, if the expressions have the same rank, that is the common rank;
- otherwise, the program is ill-formed.

(Determination of common rank is commutative and associative; the common rank of more than two expressions can be determined by arbitrarily pairing expressions and intermediate results.)

The rank of a section expression (*postfix-expression* [*section-triplet*]) is one greater than the rank of its postfix expression operand. The rank of each expression in a section triplet shall be zero.

The rank of a simple subscript expression (*postfix-expression* [*expression*]) is the sum of the ranks of its operand expressions. The rank of the subscript operand shall not be greater than one.

The rank of an argument expression list (in a function-call expression) is the common rank of the argument expressions if there are more than one, or the rank of the expression if there is exactly one, or zero if there are no expressions.

The rank of a non-member function-call expression is the rank of its argument expression list. The rank of the postfix expression identifying the function to call shall be zero.

The rank of a member function call expression is determined as if the object expression appeared as an additional expression in the argument list.

The rank of a comma expression is the rank of its second operand.

The rank of a *lambda-expression* is zero.

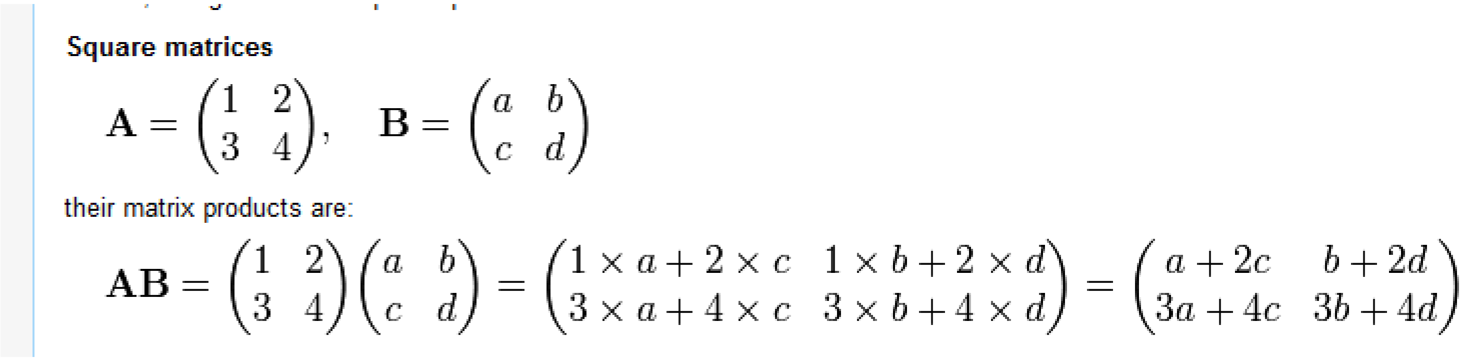
In an assignment expression, if the right operand has nonzero rank, the left operand shall have the same rank as the right operand.

Examples of Rank

***Rank** with no further specifications is usually a synonym for (or refers to) "number of dimensions";*

Source: http://en.wikipedia.org/wiki/Rank_of_an_array

Expression	Rank
A[3:4][0:10]	2
A[3][0:10]	1
A[3:4][0]	1
A[:][:]	2
A[3][0]	0



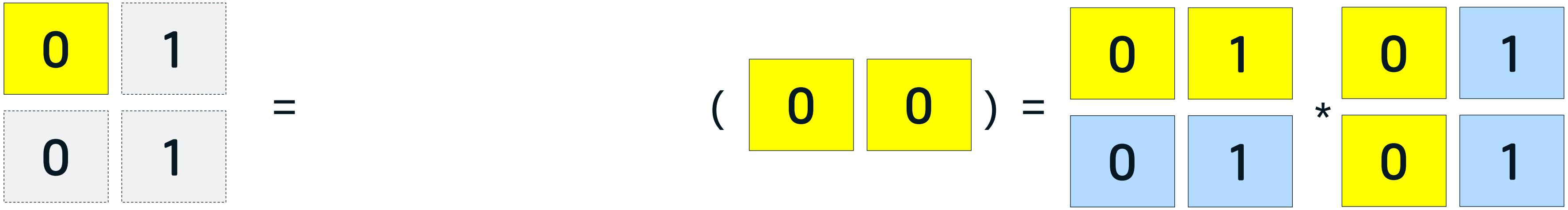
Square matrices

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

their matrix products are:

$$\mathbf{AB} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} 1\times a + 2\times c & 1\times b + 2\times d \\ 3\times a + 4\times c & 3\times b + 4\times d \end{pmatrix} = \begin{pmatrix} a + 2c & b + 2d \\ 3a + 4c & 3b + 4d \end{pmatrix}$$

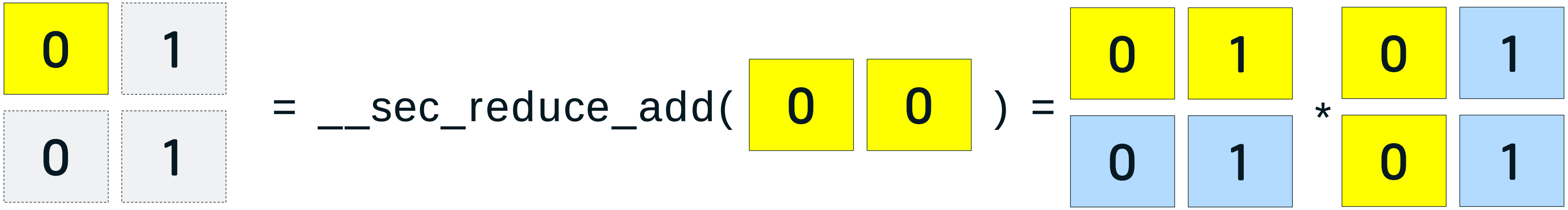
Source: http://en.wikipedia.org/wiki/Matrix_multiplication



$$\begin{matrix} 0 & 1 \\ 0 & 1 \end{matrix} =$$

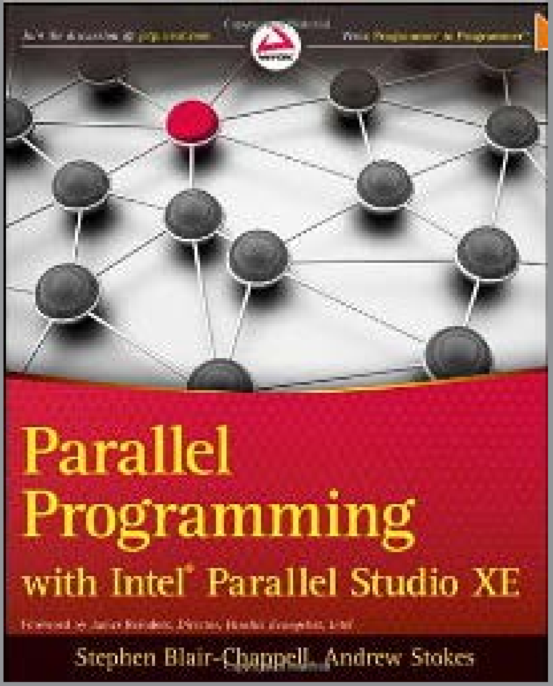
$$(\; 0 \quad 0 \;) = \begin{matrix} 0 & 1 \\ 0 & 1 \end{matrix} * \begin{matrix} 0 & 1 \\ 0 & 1 \end{matrix}$$

```
for (i = 0; i < N; i++)
  {
    for (j = 0; j < N; j++)
    {
      c[ i][ j] __sec_reduce_add = a[ i][:] * b[:][ j];
    }
  }
```



$$\begin{matrix}0&1\\0&1\end{matrix} = \texttt{\_\_sec\_reduce\_add(}\ \begin{matrix}0&0\end{matrix}\ \texttt{)} = \begin{matrix}0&1\\0&1\end{matrix} * \begin{matrix}0&1\\0&1\end{matrix}$$

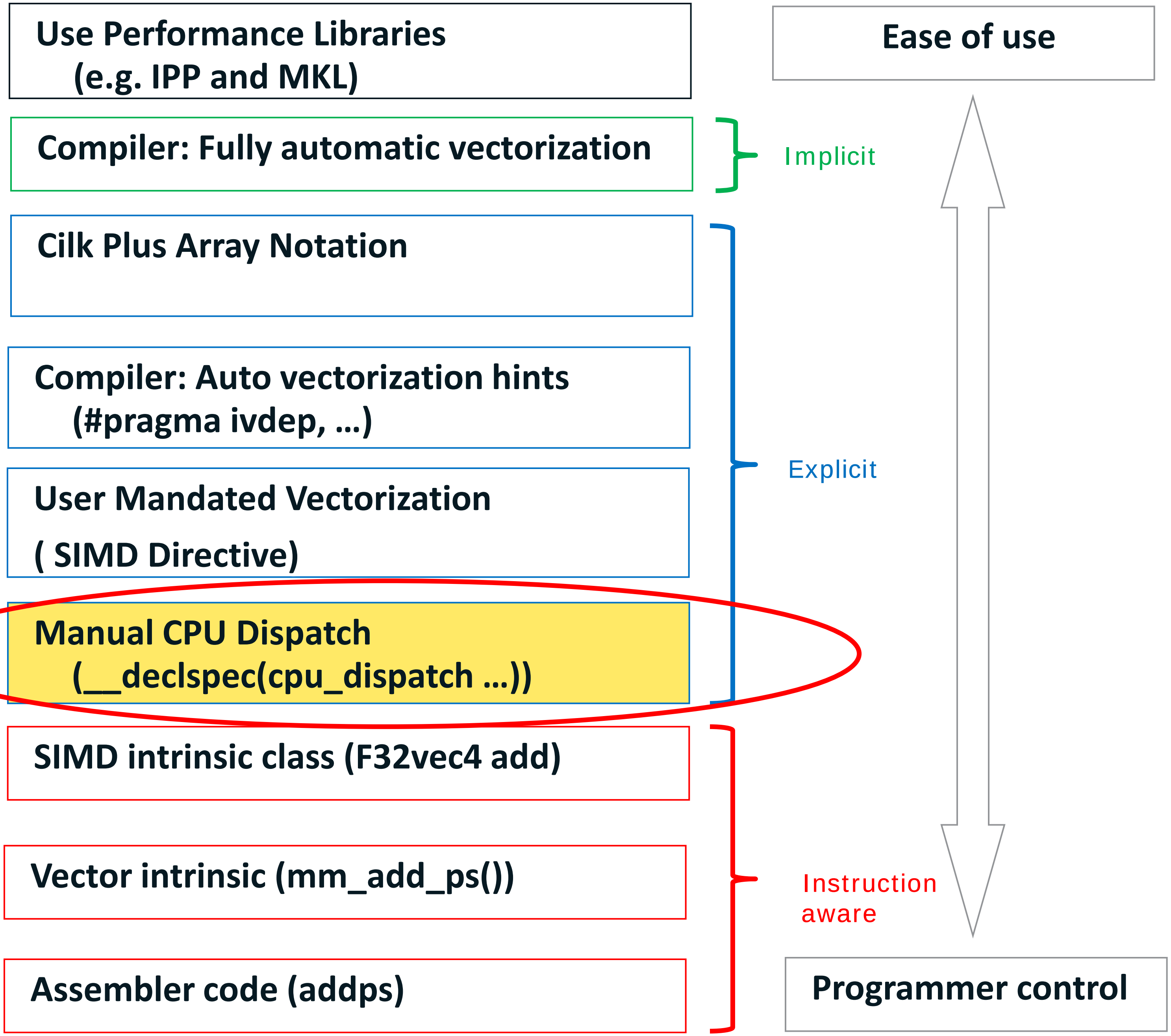
Hands-on Lab



Explicit Vectorisation

Cilk Plus array Notation
Elemental Functions

Other Ways of Inserting Vectorised Code



Ease of use

Use Performance Libraries (e.g. IPP and MKL)

Implicit
- Compiler: Fully automatic vectorization

Explicit
- Cilk Plus Array Notation
- Compiler: Auto vectorization hints (#pragma ivdep, …)
- User Mandated Vectorization (SIMD Directive)
- **Manual CPU Dispatch (__declspec(cpu_dispatch …))**

Instruction aware
- SIMD intrinsic class (F32vec4 add)
- Vector intrinsic (mm_add_ps())
- Assembler code (addps)

Programmer control

Manual processor Dispatch

Allows you to write processor-specific code

Provide more than one version of code

Use __declespec(cpu_dispatch(cpuid,cpuid...)

CPUID Arguments

Argument for cpuid	Processors
future_cpu_16 (subject to change)	2nd generation Intel® Core™ processor family with support for Intel® Advanced Vector Extensions (Intel® AVX).
core_aes_pclmulqdq	Intel® Core™ processors with support for Advanced Encryption Standard (AES) instructions and carry-less multiplication instruction
core_i7_sse4_2	Intel® Core™ processor family with support for Intel® SSE4 Efficient Accelerated String and Text Processing instructions (SSE4.2)
atom	Intel® Atom™ processors
core_2_duo_sse4_1	Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture processors with support for Intel® SSE4 Vectorizing Compiler and Media Accelerators instructions (SSE4.1)
core_2_duo_ssse3	Intel® Core™2 Duo processors and Intel® Xeon® processors with Intel® Supplemental Streaming SIMD Extensions 3 (SSSE3)
pentium_4_sse3	Intel® Pentium 4 processor with Intel® Streaming SIMD Extensions 3 (Intel® SSE3), Intel® Core™ Duo processors, Intel® Core™ Solo processors
pentium_4	Intel® Intel Pentium 4 processors
pentium_m	Intel® Pentium M processors
pentium_iii	Intel® Pentium III processors
generic	Other IA-32 or Intel 64 processors or compatible processors not provided by Intel Corporation

Manual Dispatch Example

```
#include <stdio.h>
 // need to create specific function versions
__declspec(cpu_dispatch(generic, future_cpu_16))
void dispatch_func() {};

__declspec(cpu_specific(generic))
void dispatch_func() {
  printf("Code for non-Intel processors\and generic Intel\n");
}

__declspec(cpu_specific(future_cpu_16))
void dispatch_func() {
  printf("Code for 2nd generation Intel Core processors goes here\n");
}
int main() {
  dispatch_func();
  printf("Return from dispatch_func\n");
  return 0;
}
```

Step 6

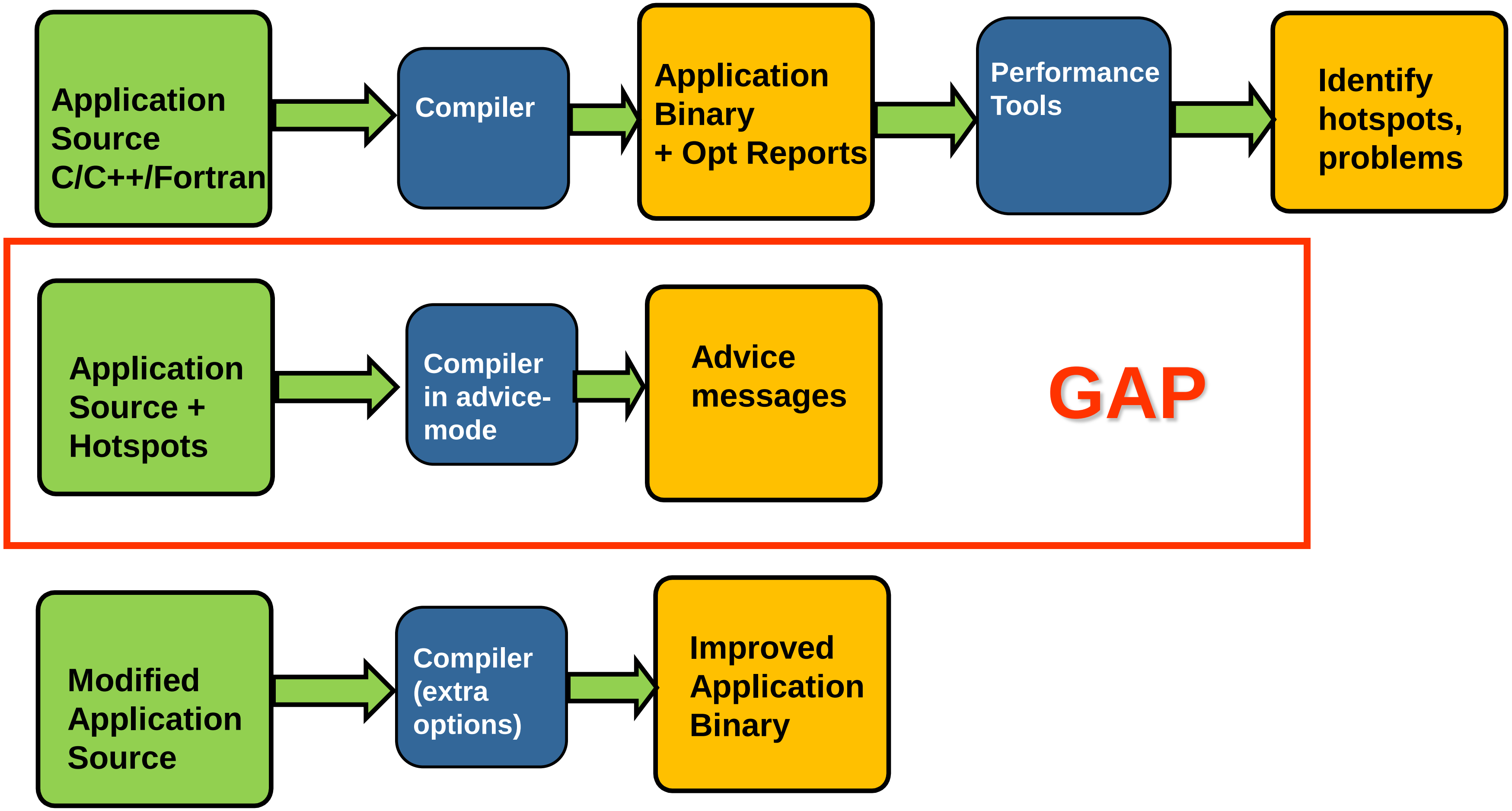
Tuning Automatic Vectorization

GAP – Guided Automatic Parallelization

Key design ideas:

- It gives Advice!
- On automatic Parallelism
- On Vectorisation
- Is *not* a code generator
- Is *not* a replacement for the other compiler reports
- Works with C/C++ and Fortran

Workflow with Compiler as a Tool



Application Source C/C++/Fortran → Compiler → Application Binary + Opt Reports → Performance Tools → Identify hotspots, problems

Application Source + Hotspots → Compiler in advice-mode → Advice messages

GAP

Modified Application Source → Compiler (extra options) → Improved Application Binary

Simplifies programmer effort in application tuning

GAP – How it Works

Compiler Switches for GAP [1]

Activate GAP and optionally define guidance level

{L&M}: -guide[=level] {W}: /Qguide[:level]

Activate GAP individually for auto-vectorization, auto-parallelization or data transformations

{L&M}:

- -guide-vec[=level]
- -guide-par[=level]
- -guide-data-trans[=level]

{W}:

- -guide-vec[=level]
- -guide-par[=level]
- -guide-data-trans[=level]

Optional argument level=1,2,3,4 controls extend of analysis: ‘4’ is most advanced / most detailed and is default

You must also specify option -parallel (Linux* OS and Mac OS* X) or /Qparallel (Windows* OS) to receive auto-parallelization guidance

GAP – How it Works

Compiler Switches for GAP [2]

Control the source code part for which analysis is done

{L&M}: `-guide-opts=<string>` {W}: `/Qguide-opts:<string>`

Samples for <string>:

-“init.c, 1-50,100-150“

Restrict analysis to file init.c, lines 1-50 and 100-150

-"bar.f90,'m1::func_solve`“

Restrict analysis to file bar.f90, Fortran module “m1”, function ‘func_solve’

Control where the message are going – into a new file or append messages to existing file

{L&M}:

`-guide-file=<file_name>`

`-guide-file-append=<file_name>`

{W}:

`/Qguide-file:<file_name>`

`/Qguide-file-append:<file_name>`

Vectorization Example [1]

```
void f(int n, float *x, float *y, float *z, float *d1, float *d2)

{

  for (int i = 0; i < n; i++)

          z[i] = x[i] + y[i] – (d1[i]*d2[i]);

}
```

GAP Message:

g.c(6): remark #30536: (LOOP) Add -Qno-alias-args option for better type-based disambiguation analysis by the compiler, if appropriate (the option will apply for the entire compilation). This will improve optimizations such as vectorization for the loop at line 6. [VERIFY] Make sure that the semantics of this option is obeyed for the entire compilation. [ALTERNATIVE] Another way to get the same effect is to add the "restrict" keyword to each pointer-typed formal parameter of the routine "f". This allows optimizations such as vectorization to be applied to the loop at line 6. [VERIFY] Make sure that semantics of the "restrict" pointer qualifier is satisfied: in the routine, all data accessed through the pointer must not be accessed through any other

The compiler guides the user on source-change and on what pragma to insert and on how to determine whether that pragma is correct for this case

Vectorization Example [2]

```
void mul(NetEnv* ne, Vector* rslt
    Vector* den, Vector* flux1,
    Vector* flux2, Vector* num
    {
      float *r, *d, *n, *s1, *s2;
      int i;
      r=rslt->data;  d=den->data;
      n=num->data; s1=flux1->data;
      s2=flux2->data;

   for (i = 0; i < ne->len; ++i)
      r[i] = s1[i]*s2[i] +
      n[i]*d[i];
    }
```

GAP Messages (simplified):

1. “Use a local variable to hoist the upper-bound of loop at line 29 (variable:ne->len) if the upper-bound does not change during execution of the loop”

2. “Use “#pragma ivdep" to help vectorize the loop at line 29, if these arrays in the loop do not have cross-iteration dependencies: r, s1, s2, n, d”

-> Upon recompilation, the loop will be vectorized

Data Transformation Example

```
struct S3 {

     int a;
    int b; // hot
    double c[100];
    struct S2 *s2_ptr;
    int d;  int e;
    struct S1 *s1_ptr;
    char *c_p;
    int f; // hot
    };
```

```
…
for (ii = 0; ii < N; ii++){
        sp->b = ii;
        sp->f = ii + 1;
        sp++;
}
…
```

```
peel.c(22): remark #30756: (DTRANS) Splitting the structure 'S3' into
    two parts will improve data locality and is highly recommended.
    Frequently accessed fields are 'b, f'; performance may improve by
    putting these fields into one structure and the remaining fields
    into another structure. Alternatively, performance may also improve
    by reordering the fields of the structure. Suggested field order:'b,
    f, s2_ptr, s1_ptr, a, c, d, e, c_p'. [VERIFY] The suggestion is
    based on the field references in current compilation …
```

Compiler Options that help Vectorisation

- -O3 (/O3) performs other loop transformations first
- -ipo (/Qipo) may inline, or get dependency, loop count or alignment information from calling functions
- -xavx (/QxAVX) use all available instructions

  -xhost (/QxHOST)

- -fno-alias (/Oa) assume pointers not aliased (dangerous!)
- -fargument-noalias assume function arguments not aliased

  (/Qalias-args-)
- -fansi-alias assume different data types not aliased

  (/Qansi-alias)
- -guide (/Qguide) get advice on how to help the compiler to vectorize loops

Thank You

Legal Disclaimer & Optimization Notice

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.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

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