Implicit Vectorisation

Stephen Blair-Chappell

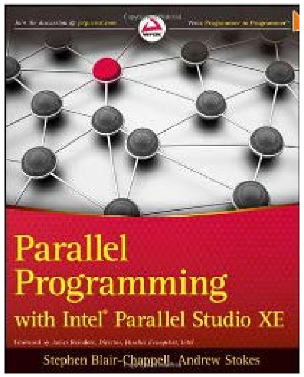
Intel Compiler Labs

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*


Parallel
Programming
with Intel® Parallel Studio XE
Stephen Blair-Chappell, Andrew Stokes

Optimization Notice

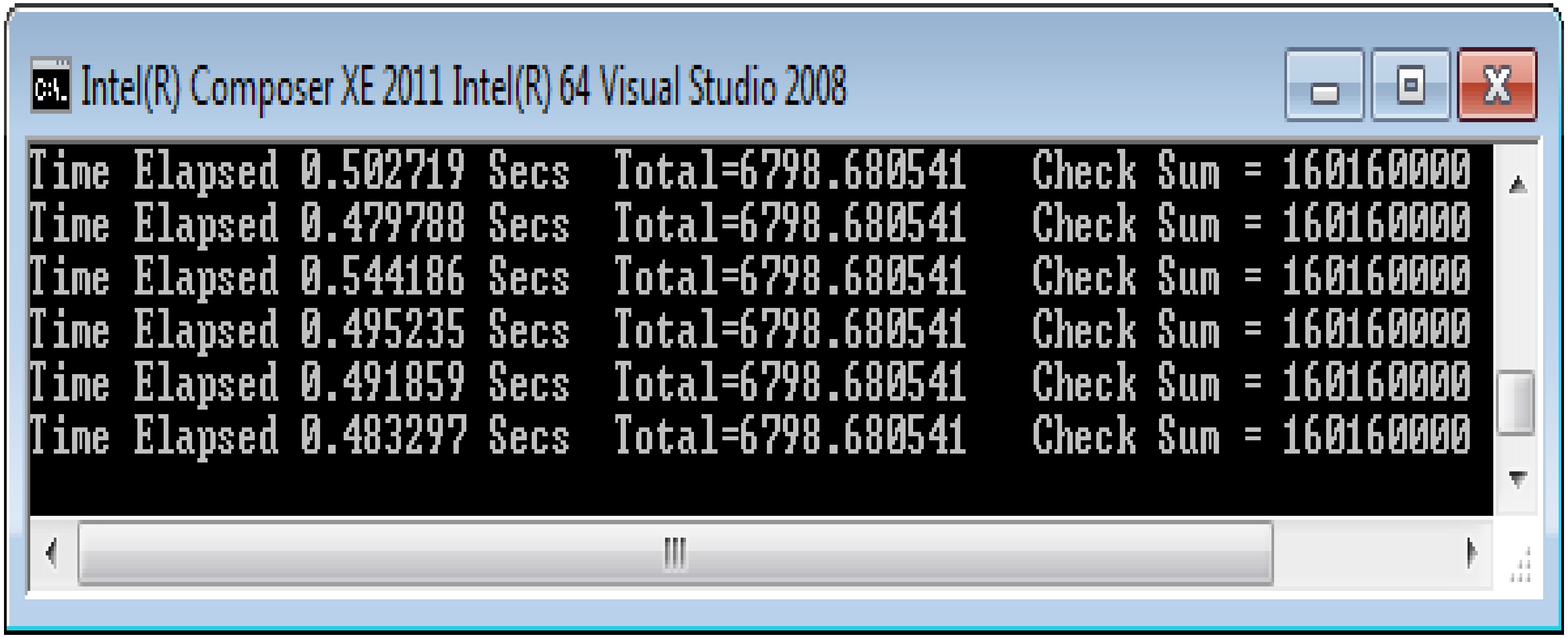
What’s in this section?

- (A seven-step optimization process)
- Using different compiler options to optimize your code
- Using auto-vectorization to tune your application to different CPUs

Optimization Notice

The Sample Application

- Initialises two matrices with a numeric sequence
- Does a Matrix Multiplication



```
Time Elapsed 0.502719 Secs  Total=6798.680541   Check Sum = 160160000
Time Elapsed 0.479788 Secs  Total=6798.680541   Check Sum = 160160000
Time Elapsed 0.544186 Secs  Total=6798.680541   Check Sum = 160160000
Time Elapsed 0.495235 Secs  Total=6798.680541   Check Sum = 160160000
Time Elapsed 0.491859 Secs  Total=6798.680541   Check Sum = 160160000
Time Elapsed 0.483297 Secs  Total=6798.680541   Check Sum = 160160000
```

The main loop (without timing & printf)

```
  // repeat experiment six times
   for( l=0; l<6; l++ )
   {
      // initialize matrix a
      sum = Work(&total,a);

      // initialize matrix b;
      for (i = 0; i < N; i++) {
        for (j=0; j<N; j++) {
          for (k=0;k<DENOM_LOOP;k++) {
            sum += m/denominator;
          }
          b[N*i + j] = sum;
        }
      }
      // do the matrix manipulation
      MatrixMul( (double (*)[N])a, (double (*)[N])b, (double (*)[N])c);
 }
```

The Matrix Multiply

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

The Seven Optimisation Steps

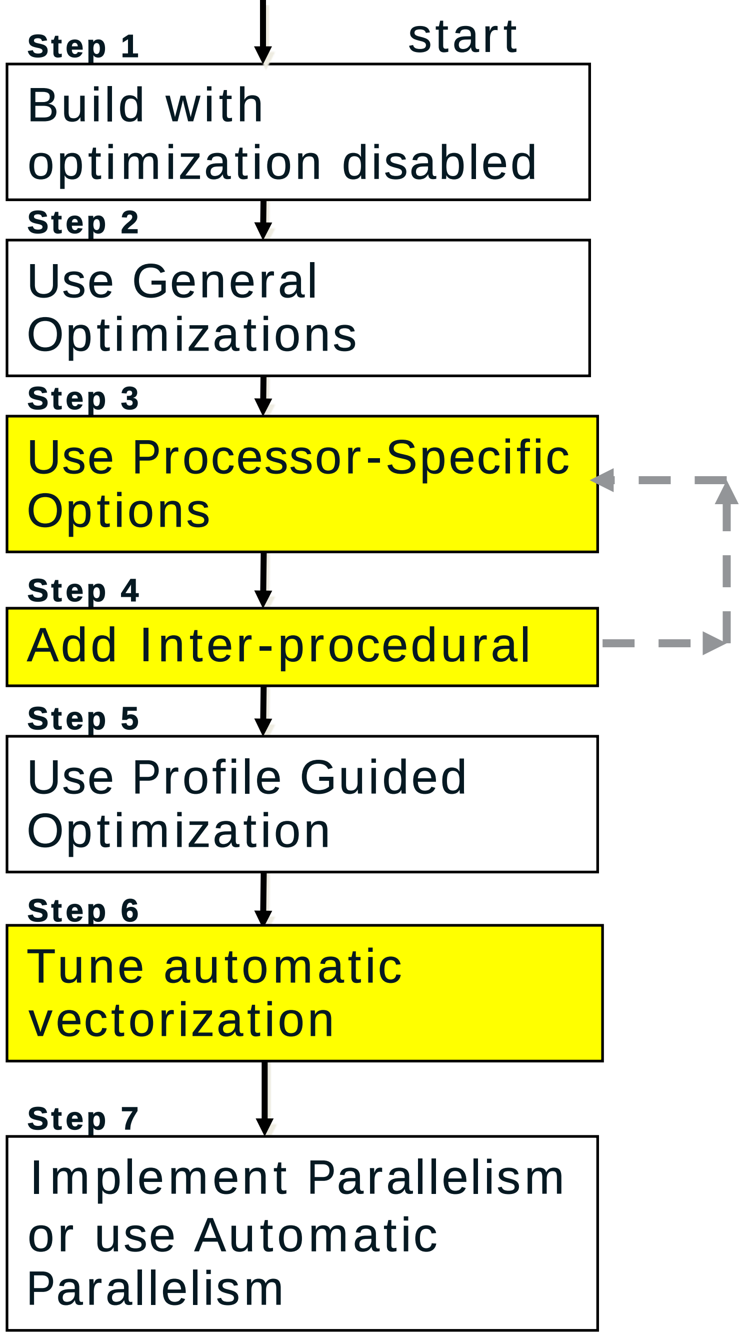
start

Example options

		Windows	*(Linux)*
Step 1	Build with optimization disabled	/Od	(-O0)
Step 2	Use General Optimizations	/01,/02,/03	(-O1, -O2, -O3)
Step 3	Use Processor-Specific Options	/QxSSE4.2 /QxHOST	(-xsse4.2) (-xhost)
Step 4	Add Inter-procedural	/Qipo	(-ipo)
Step 5	Use Profile Guided Optimization	/Qprof-gen /Qprof-use	(-prof-gen) (-prof-use)
Step 6	Tune automatic vectorization	/Qguide	(-guide)
Step 7	Implement Parallelism or use Automatic Parallelism	Use Intel Family of Parallel Models /Qparallel	(-parallel)

Optimization Notice

The Seven Optimisation Steps


start
Step 1
Build with optimization disabled
Step 2
Use General Optimizations
Step 3
Use Processor-Specific Options
Step 4
Add Inter-procedural
Step 5
Use Profile Guided Optimization
Step 6
Tune automatic vectorization
Step 7
Implement Parallelism or use Automatic Parallelism

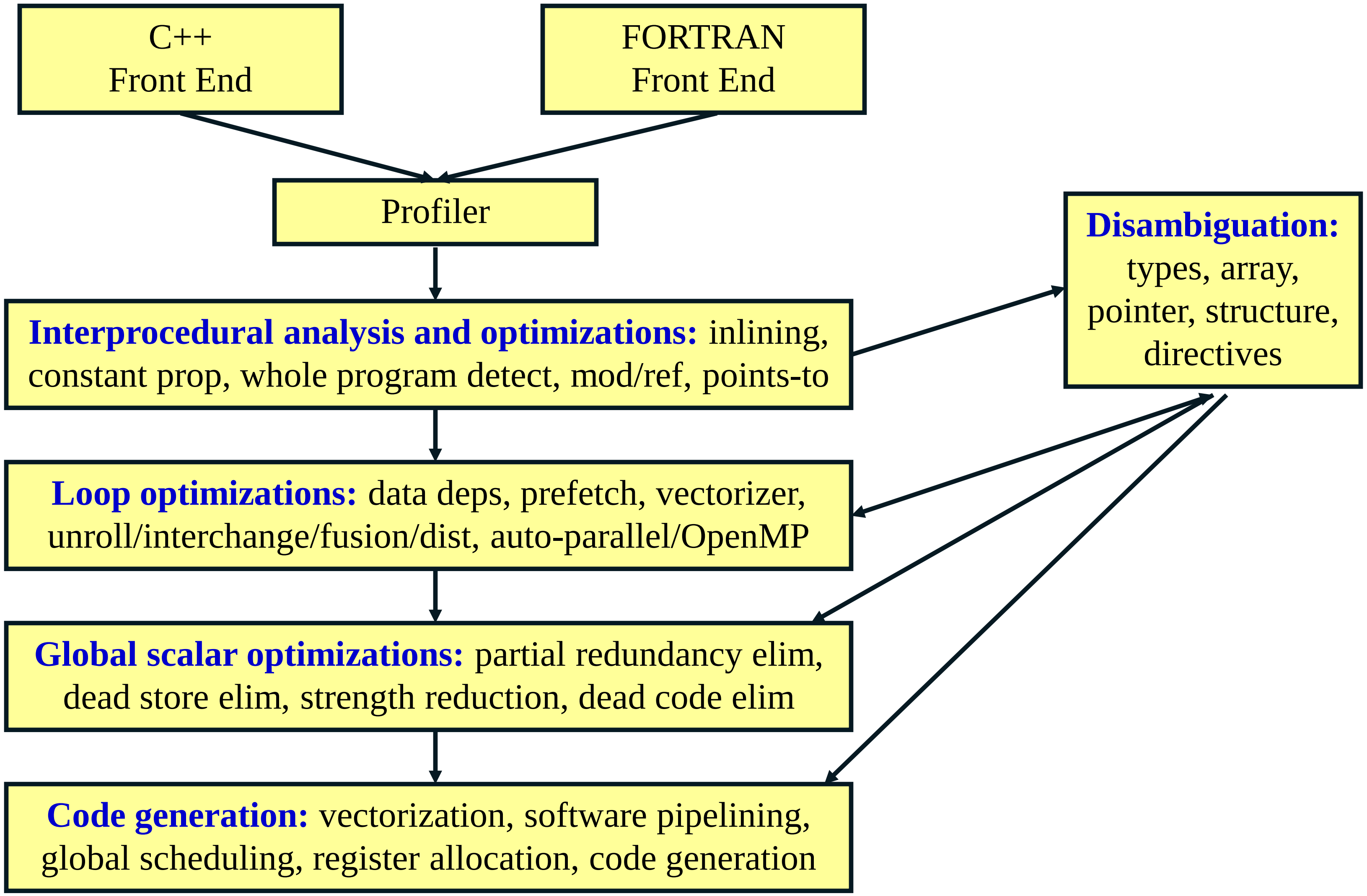
Example options

Windows	*(Linux)*
/Od	(-O0)
/01,/02,/03	(-O1, -O2, -O3)
/QxSSE4.2	(-xsse4.2)
/QxHOST	(-xhost)
/Qipo	(-ipo)
/Qprof-gen	(-prof-gen)
/Qprof-use	(-prof-use)
/Qguide	(-guide)

Use Intel Family of Parallel Models

/Qparallel	(-parallel)

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: types, array, pointer, structure, directives

Step 2

Optimization Notice

Getting Visibility : Compiler Optimization Report

Compiler switch:

`-opt-report-phase[=phase]` (Linux)

‚phase‘ can be:

- `ipo` – Interprocedural Optimization
- `ilo` – Intermediate Language Scalar Optimization
- `hpo` – High Performance Optimization
- `hlo` – High-level Optimization

…

- `all` – All optimizations (not recommended, output too verbose)

Control the level of detail in the report:

`/Qopt-report[0|1|2|3]` (Windows)

`-opt-report[0|1|2|3]` (Linux, MacOS X)

Step 2

Optimization Report Example

```
icc –O3 –opt-report-phase=hlo -opt-report-phase=hpo
icl /O3 /Qopt-report-phase:hlo /Qopt-report-phase:hpo
```

```
…
LOOP INTERCHANGE in loops at line: 7 8 9
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )
…
Loop at line 8 blocked by 128
Loop at line 9 blocked by 128
Loop at line 10 blocked by 128
…
Loop at line 10 unrolled and jammed by 4
Loop at line 8 unrolled and jammed by 4
…
…(10)…  loop was not vectorized: not inner loop.
…(8)…   loop was not vectorized: not inner loop.
…(9)…   PERMUTED LOOP WAS VECTORIZED
…
```

icc –vec-report2 (icl /Qvec-report2) for just the vectorization report

Step 2

Optimization Notice

There are lots of Phases!

```
icl /Qopt-report-help
Intel(R) C++ Intel(R) 64 Compiler XE for applications running on Intel(R)
64, Version 12.0.3.175 Build 20110309
Copyright (C) 1985-2011 Intel Corporation.  All rights reserved.
Intel(R) Compiler Optimization Report Phases
usage:  -Qopt_report_phase <phase>
ipo, ipo_inl, ipo_cp, ipo_align, ipo_modref, ipo_lpt, ipo_subst,
ipo_ratt, ipo_vaddr, ipo_pdce, ipo_dp, ipo_gprel, ipo_pmerge,
ipo_dstat, ipo_fps, ipo_ppi, ipo_unref, ipo_wp, ipo_dl,
ipo_psplit, ilo, ilo_arg_prefetching, ilo_lowering,
ilo_strength_reduction, ilo_reassociation, ilo_copy_propagation,
ilo_convert_insertion, ilo_convert_removal, ilo_tail_recursion,
hlo, hlo_fusion, hlo_distribution, hlo_scalar_replacement,
hlo_unroll, hlo_prefetch, hlo_loadpair, hlo_linear_trans,
hlo_opt_pred, hlo_data_trans, hlo_string_shift_replace, hlo_ftae,
hlo_reroll, hlo_array_contraction, hlo_scalar_expansion,
hlo_gen_matmul, hlo_loop_collapsing, hpo, hpo_analysis,
hpo_openmp, hpo_threadization, hpo_vectorization, pgo, tcollect,
offload, all
```

Step 2

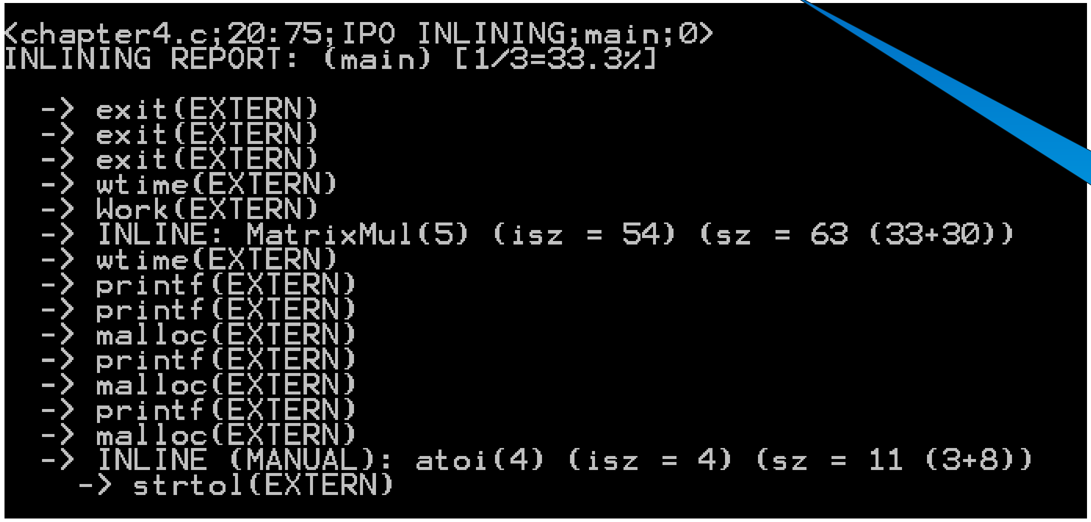
Optimization Notice

Getting Visibility : Assembler Listing

make chapter4.o CFLAGS="-O2 -opt-report -S"

Generate assembler file

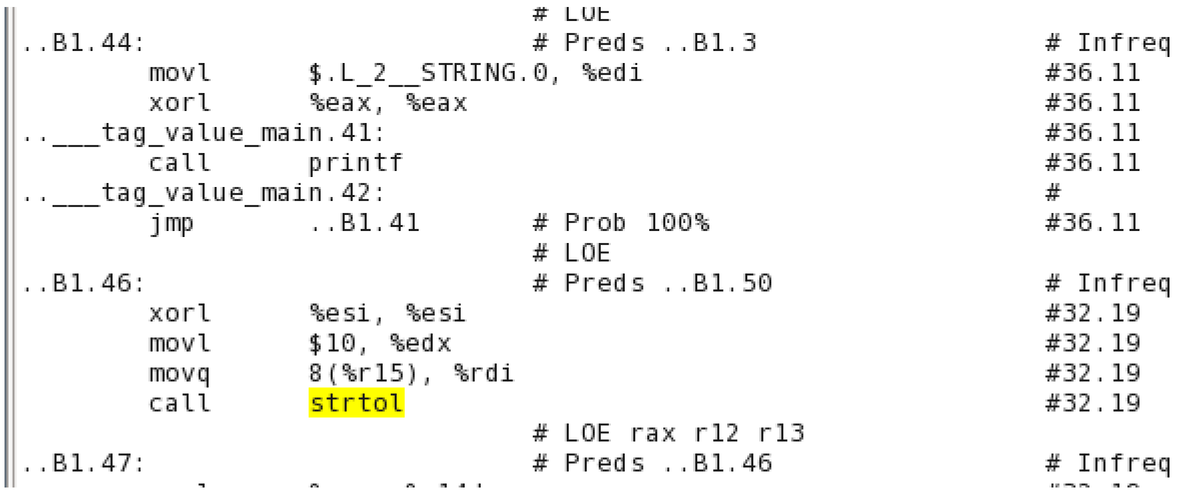
Create a report



```
<chapter4.c;20:75;IPO INLINING;main;0>
INLINING REPORT: (main) [1/3=33.3%]

  -> exit(EXTERN)
  -> exit(EXTERN)
  -> exit(EXTERN)
  -> wtime(EXTERN)
  -> Work(EXTERN)
  -> INLINE: MatrixMul(5) (isz = 54) (sz = 63 (33+30))
  -> wtime(EXTERN)
  -> printf(EXTERN)
  -> printf(EXTERN)
  -> malloc(EXTERN)
  -> printf(EXTERN)
  -> malloc(EXTERN)
  -> printf(EXTERN)
  -> malloc(EXTERN)
  -> INLINE (MANUAL): atoi(4) (isz = 4) (sz = 11 (3+8))
    -> strtol(EXTERN)
```

Assembler Code



```
                                # LOE
..B1.44:                        # Preds ..B1.3                  # Infreq
        movl      $.L_2__STRING.0, %edi                         #36.11
        xorl      %eax, %eax                                    #36.11
..___tag_value_main.41:                                         #36.11
        call      printf                                        #36.11
..___tag_value_main.42:                                         #
        jmp       ..B1.41       # Prob 100%                     #36.11
                                # LOE
..B1.46:                        # Preds ..B1.50                 # Infreq
        xorl      %esi, %esi                                    #32.19
        movl      $10, %edx                                     #32.19
        movq      8(%r15), %rdi                                 #32.19
        call      strtol                                        #32.19
                                # LOE rax r12 r13
..B1.47:                        # Preds ..B1.46                 # Infreq
```

Source Code

```
if(argc == 2)
  denominator = atoi(argv[1]);
```

Step 2

Optimization Notice

Step 3

Using Processor Specific Options

Optimization Notice

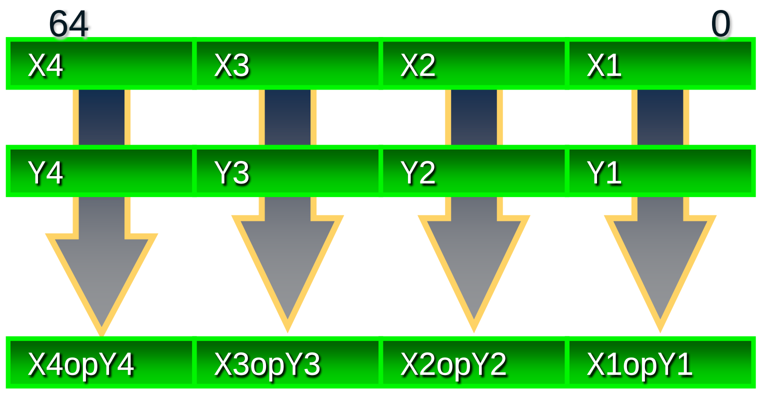
SIMD Instruction Enhancements

Year	Extension	Details
1999	SSE	70 instr Single-Precision Vectors Streaming operations
2000	SSE2	144 instr Double-precision Vectors 8/16/32 64/128-bit vector integer
2004	SSE3	13 instr Complex Data
2006	SSSE3	32 instr Decode
2007	SSE4.1	47 instr Video Graphics building blocks Advanced vector instr
2008	SSE4.2	8 instr String/XML processing POP-Count CRC
2009	AES-NI	7 instr Encryption and Decryption Key Generation
2011	AVX	~100 new instr. ~300 legacy sse instr updated 256-bit vector 3 and 4-operand instructions
2012\2013	AVX2	Int. AVX expands to 256 bit Improved bit manip. fma Vector shifts Gather
2012	MIC	512-bit vector

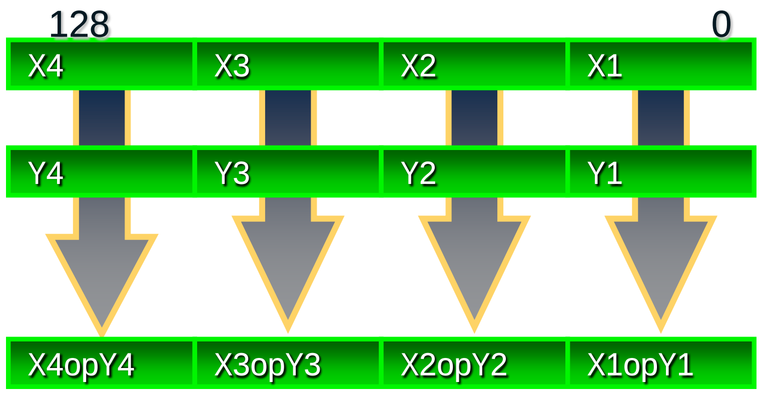
Step 3

Optimization Notice

SIMD Types in Processors from Intel [1]

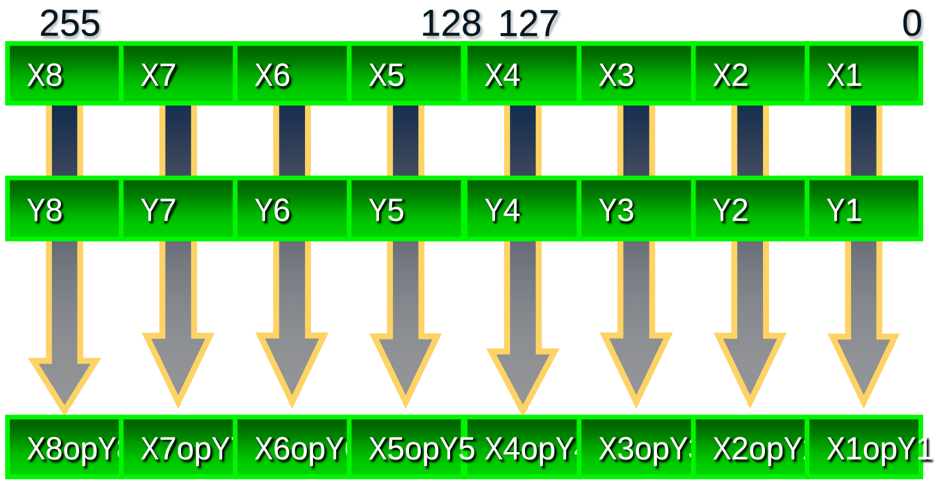


MMX™
Vector size: 64bit
Data types: 8, 16 and 32 bit integers
VL: 2,4,8
For sample on the left: Xi, Yi 16 bit integers



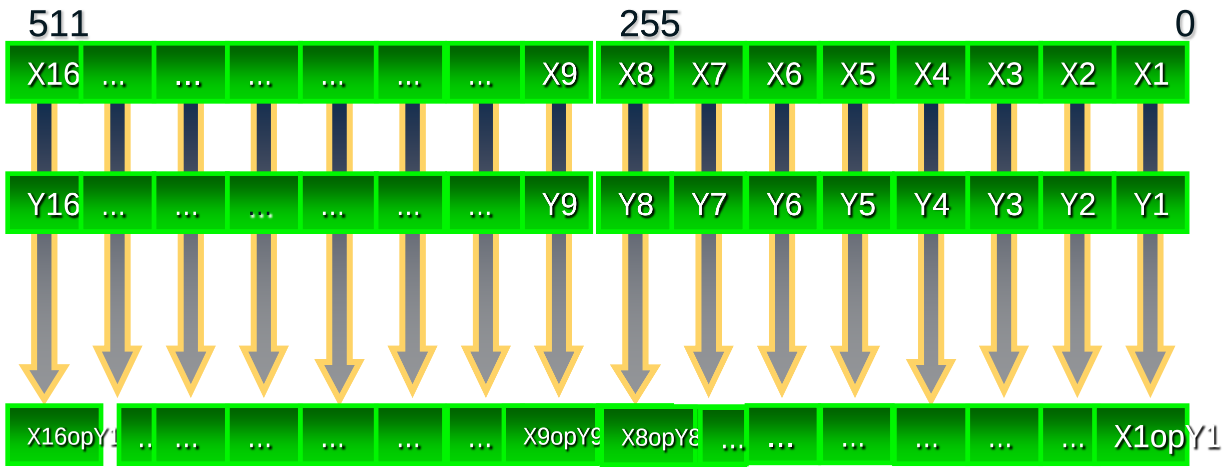
Intel® SSE
Vector size: 128bit
Data types:
  8,16,32,64 bit integers
  32 and 64bit floats
VL: 2,4,8,16
Sample: Xi, Yi bit 32 int / float

SIMD Types in Processors from Intel [2]



Intel® AVX

Vector size: 256bit
Data types: 32 and 64 bit floats
VL: 4, 8, 16
Sample: Xi, Yi 32 bit int or float

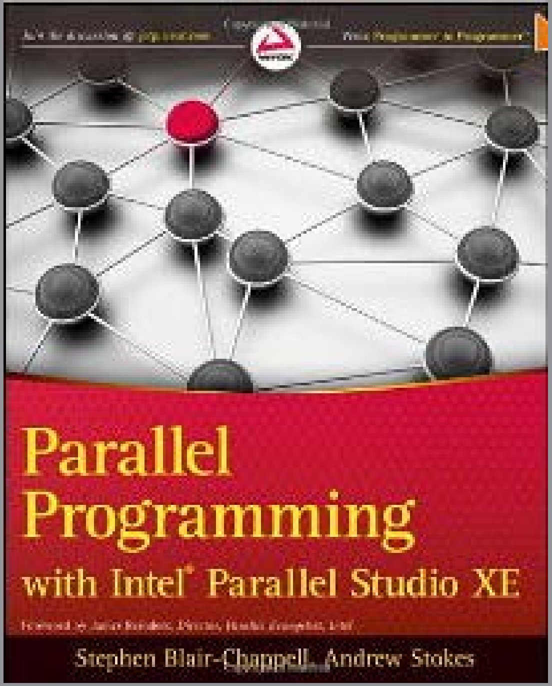

511
255
0
X16 ... ... X9 X8 X7 X6 X5 X4 X3 X2 X1
Y16 ... ... Y9 Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1
X16opY16 ... ... X9opY9 X8opY8 ... ... X1opY1

Intel® MIC

Vector size: 512bit
Data types:
  32 and 64 bit integers
  32 and 64bit floats
  (some support for
    16 bits floats)
VL: 8,16
Sample: 32 bit float

Optimization Notice

Hands-on Lab


Parallel Programming with Intel Parallel Studio XE
Stephen Blair-Chappell, Andrew Stokes

Activity 4-1(plus)

Building the Application and getting a report

Key Intel® Advanced Vector Extensions (Intel® AVX) Features

KEY FEATURES	BENEFITS
• Wider Vectors – Increased from 128 to 256 bit – Two 128-bit load ports	• Up to 2x peak FLOPs (floating point operations per second) output with good power efficiency
• Enhanced Data Rearrangement – Use the new 256 bit primitives to broadcast, mask loads and permute data	• Organize, access and pull only necessary data more quickly and efficiently
• Three and four Operands: Non Destructive Syntax for both AVX 128 and AVX 256	• Fewer register copies, better register use for both vector and scalar code
• Flexible unaligned memory access support	• More opportunities to fuse load and compute operations
• Extensible new opcode (VEX)	• Code size reduction

Intel® AVX is a general purpose architecture, expected to supplant SSE in all applications used today

A New 3- and 4- Operand Instruction Format

- Intel® Advanced Vector Extensions (Intel® AVX) has a distinct destination argument that results in fewer register copies, better register use, more load/op macro-fusion opportunities, and smaller code size

xmm10 = xmm9 + xmm1

```
movaps xmm10, xmm9
addpd xmm10, xmm1
```

```
vaddpd xmm10, xmm9, x
```

1 less copy,
3 bytes smaller code size

xmm10 = xmm9 + m128

```
movups xmm10, m128
addpd xmm10, xmm9
```

```
vaddpd xmm10, xmm9,
```

1 more load/op
fusion opportunity,
4+ bytes smaller
code size

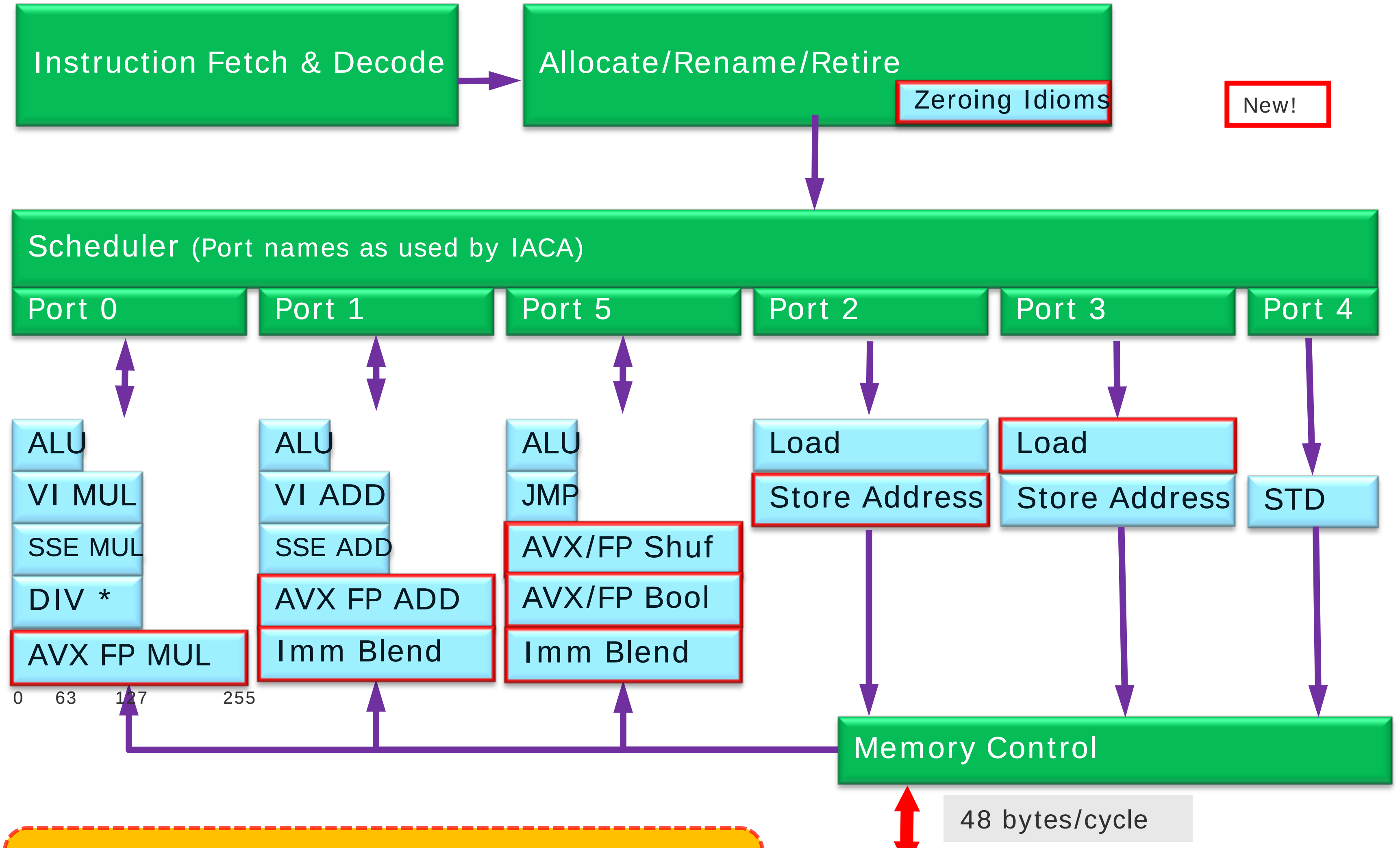
- New 4- operand Blends example, implicit xmm0 not longer needed

```
movaps xmm0, xmm4
movaps xmm1, xmm2
blendvps xmm1, m128
```

```
vblendvps xmm1, xmm2, m128, xmm4
```

Optimization Notice

Intel® Microarchitecture (Sandy Bridge) Highlights



- 1-per-cycle 256-bit multiply, add, and shuffle
- Load double the data with Intel microarchitecture (Sandy Bridge) !!!

* Not fully pipelined

Two Key Decisions to be Made :

1. How do we introduce the vector code ?

2. How do we deal with the multiple SIMD instruction set extensions like SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX ...?

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

Overview of Writing Vector Code

Array Notation

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

Elemental Function

```
__declspec(vector)
float ef(float a, float b) {
  return a + b;
}

A[:] = ef(B[:], C[:]);
```

SIMD Directive

```
#pragma simd
for (int i = 0; i < N; ++i) {
  A[i] = B[i] + C[i];
}
```

Auto-Vectorization

```
for (int i = 0; i < N; ++i) {
  A[i] = B[i] + C[i];
}
```

Step 3

Optimization Notice

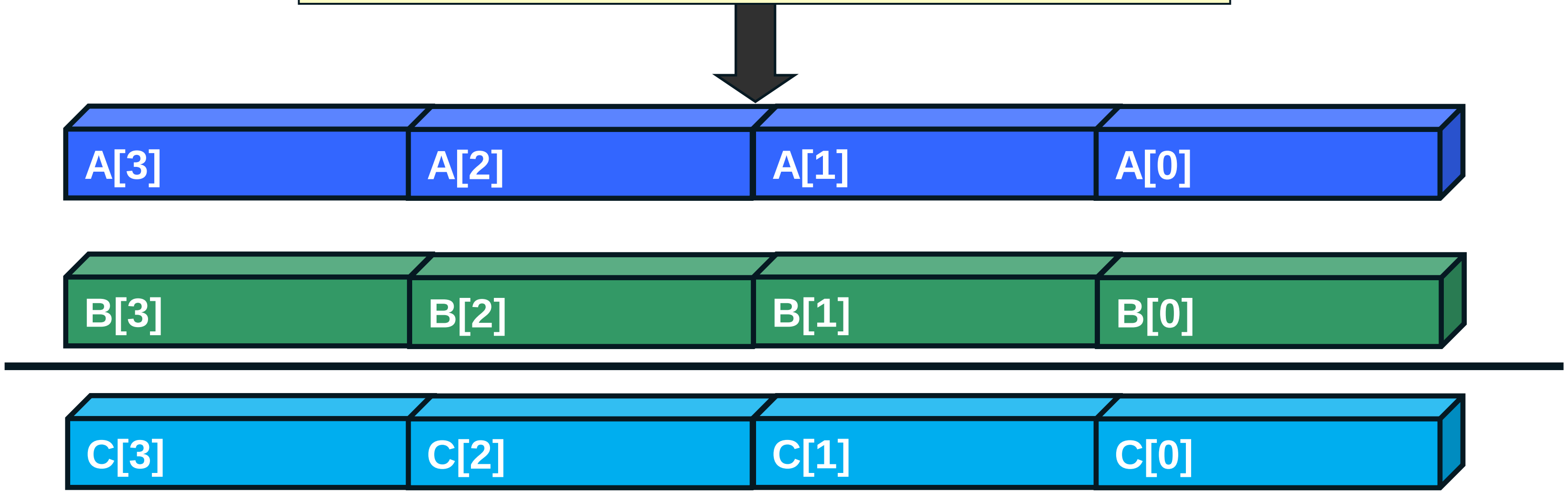
Other Ways of Inserting Vectorised Code

Use Performance Libraries (e.g. IPP and MKL)		Ease of use
Compiler: Fully automatic vectorization	Implicit	↑
Cilk Plus Array Notation	Explicit	↕
Compiler: Auto vectorization hints (#pragma ivdep, …)	Explicit	↕
User Mandated Vectorization (SIMD Directive)	Explicit	↕
Manual CPU Dispatch (__declspec(cpu_dispatch …))	Explicit	↕
SIMD intrinsic class (F32vec4 add)	Instruction aware	↕
Vector intrinsic (mm_add_ps())	Instruction aware	↓
Assembler code (addps)	Instruction aware	Programmer control

Auto-Vectorization

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

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



Step 3

How do I know if a loop is vectorised?

- -vec-report

```
> icl /Qvec-report MultArray.c
MultArray.c(92): (col. 5) remark: LOOP WAS VECTORIZED.
```

Qvec-report1 (default)

Qvec-report2

Qvec-report3

Qvec-report4

Qvec-report5

Qvec-report6

Step 3

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

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

Note:

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

Optimization Notice

How do I know if a loop is vectorised?

- -vec-report7
  - Experimental Feature
  - see http://software.intel.com/en-us/articles/vecanalysis-python-script-for-annotating-intelr-compiler-vectorization-report
  - Requires two python scripts

➢ `icc -c -vec-report7 satSub.c 2>&1 | ./vecanalysis/vecanalysis.py –list`

Message	Count	%
scalar loop cost: 3.	115	90.6%
loop was not vectorized: 1.	106	83.5%
unmasked unaligned unit stride stores: 2.	97	76.4%
heavy-overhead vector operations: 4.	84	66.1%
unmasked unaligned unit stride loads: 2.	79	62.2%
lightweight vector operations: 2.	74	58.3%
estimated potential speedup: 0.690000.	71	55.9%
vector loop cost: 4.250000.	71	

Step 3

How do I know if a loop is vectorised?

```
328: TMP1=KGLN
329: TMP2=KST
VECRPT (col. 1) LOOP WAS VECTORIZED.
VECRPT (col. 1) estimated potential speedup: 2.860000.
VECRPT (col. 1) lightweight vector operations: 17.
VECRPT (col. 1) loop inside vectorized loop at nesting level: 1.
VECRPT (col. 1) loop was vectorized (with peel/with remainder)
VECRPT (col. 1) medium-overhead vector operations: 4.
VECRPT (col. 1) remainder loop was not vectorized: 1.
VECRPT (col. 1) scalar loop cost: 7.
VECRPT (col. 1) unmasked aligned unit stride stores: 2.
VECRPT (col. 1) unmasked unaligned unit stride loads: 3.
VECRPT (col. 1) unmasked unaligned unit stride stores: 1.
VECRPT (col. 1) vector loop cost: 2.250000.
330: DO JLAT=TMP1,KGLX
331: !DO JLAT=KGLN,KGLX
332:   IADDR(JLAT)=KSTABUF(JLAT)+YDSL%NASLB1*(0-KFLDN)+YDSL%NASLB1*(1-KSLEV)
333: ENDDO
334:
```

Step 3

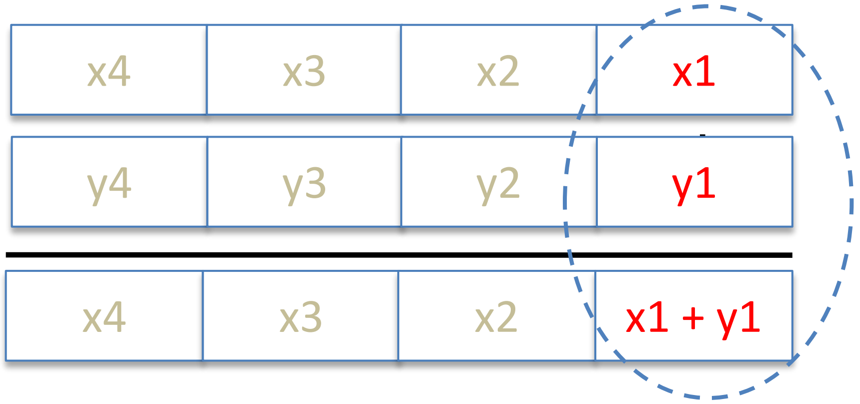
Optimization Notice

Scalar and Packed Instructions

addss **Scalar Single-FP Add**

Single precision FP data

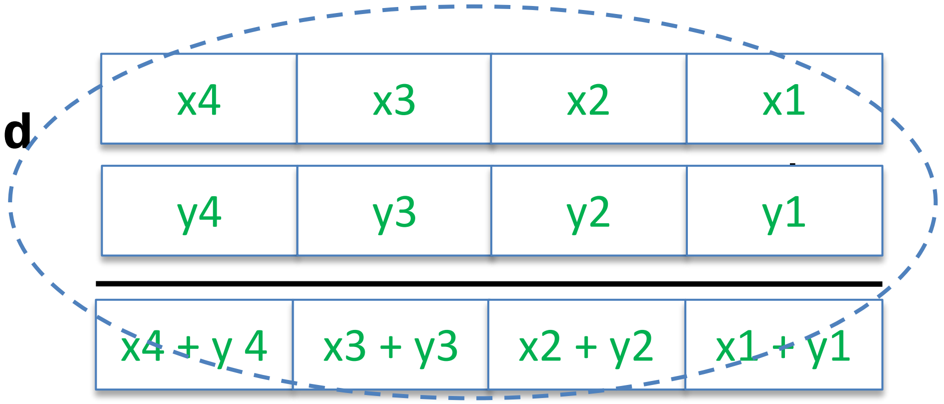
Scalar execution mode



addps **Packed Single-FP Add**

Single precision FP data

packed execution mode



Step 3

Optimization Notice

Examples of Code Generation

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

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

SSE2

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

AVX

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

SSE4.1

Step 3

Optimization Notice

Out-of-the-box behaviour – Intel Compiler

Automatic-vectorisation is enabled by default

(turn it off with –no-vec or /Qvec-)

The option –msse2 or /arch:sse2 is used by default (as long as no x, ax or –m option has been used)

-msse2: *"May generate Intel® SSE2 and SSE instructions … This value is only available on Linux systems".*

Building for non-intel CPUs /arch: (-m)

Option	Description
mic	MIC (linux only at moment)
avx	AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2, and SSE.
sse4.2	SSE4.2 SSE4.1, SSSE3, SSE3, SSE2, and SSE.
sse4.1	SSE4.1, SSSE3, SSE3, SSE2, and SSE instructions.
ssse3	SSSE3, SSE3, SSE2, and SSE instructions.
sse2	May generate Intel® SSE2 and SSE instructions.
sse	This option has been deprecated; it is now the same as specifying ia32.
ia32	Generates x86/x87 generic code that is compatible with IA-32 architecture.

This option tells the compiler to generate code specialized for the processor that executes your program.
Code generated with these options should execute on any compatible, non-Intel processor with support for the corresponding instruction set.

Step 3

Optimization Notice

Building for Intel processors /Qx (-x)

Option	Description
CORE-AVX2	AVX2, AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2, and SSE instructions .
CORE-AVX-I	RDND instr, AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2, and SSE instructions .
AVX	AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2, and SSE instructions .
SSE4.2	SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core™ i7 processors. SSE4 .1, SSSE3, SSE3, SSE2, and SSE. May optimize for the Intel® Core™ processor family.
SSE4.1	SSE4 Vectorizing Compiler and Media Accelerator, SSSE3, SSE3, SSE2, and SSE . May optimize for Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture.
SSSE3_ATOM (sse3_ATOM depracted)	MOVBE , (depending on -minstruction), SSSE3, SSE3, SSE2, and SSE . Optimizes for the Intel® Atom™ processor and Intel® Centrino® Atom™ Processor Technology
SSSE3	SSSE3, SSE3, SSE2, and SSE. Optimizes for the Intel® Core™ microarchitecture.
SSE3	SSE3, SSE2, and SSE. Optimizes for the enhanced Pentium® M processor microarchitecture and Intel NetBurst® microarchitecture.
SSE2	SSE2 and SSE . Optimizes for the Intel NetBurst® microarchitecture.

Step 3

Optimization Notice

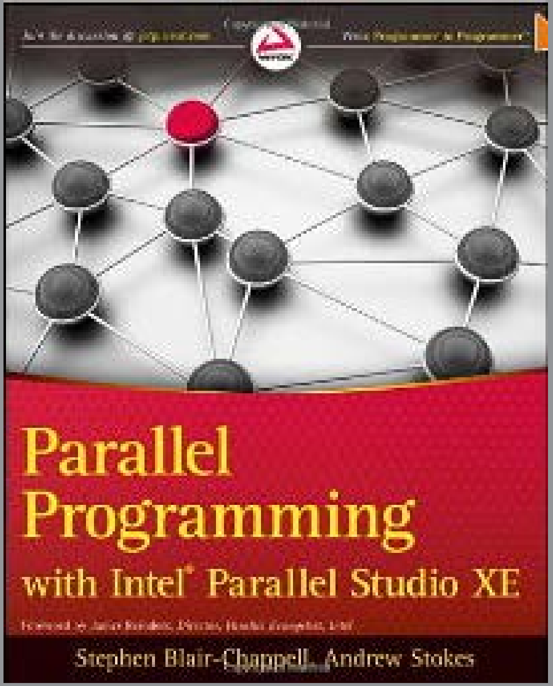
Results of Enhancing Auto-Vectorisation

SETTING	TIME	SPEEDUP
SSE2	0.293	1
AVX	0.270	1.09

Step 3

Optimization Notice

Hands-on Lab


Parallel Programming with Intel Parallel Studio XE
Stephen Blair-Chappell, Andrew Stokes

Activity 4-2

Proving that the code has been vectorised

Vectorization Report

“Loop was not vectorized” because:

- “Existence of vector dependence”
- “Non-unit stride used”
- “Mixed Data Types”
- “Condition too Complex”
- “Condition may protect exception”
- “Low trip count”
- “Subscript too complex”
- ‘Unsupported Loop Structure”
- “Contains unvectorizable statement at line XX”
- “Not Inner Loop”
- "vectorization possible but seems inefficient"
- “Operator unsuited for vectorization”

e.g. function calls

Step 3

Optimization Notice

Ways you can help the auto-vectoriser

- Change data layout – avoid non-unit strides
- Use #pragma ivdep
- Use the restrict key word (C \C++)
- Use #pragma vector always
- Use #pragma simd
- Use elemental functions
- Use array notation

Step 3

Optimization Notice

Consistency of SIMD results

Two issues can effect reproducibility

- **Alignment**
- **Parallelism**

Reason: The order the calculations are done can change

Alignment of Data

SSE2 : works better with 16 byte alignment.

Why? : the XMM registers are 16 bytes (ie 128 bits)

Penalites:

Unaligned access vs aligned access (but still in same cache line) 40% worse.

Unaligned access vs aligned access (but split over cache line) 500% worse.

Rule of Thumb: Try to align to the SIMD register size

MMX: 8 Bytes;
SSE2: 16 bytes,
AVX: 32 bytes

ALSO: Try to align blocks of data to cacheline size – ie 64 bytes

Source: http://software.intel.com/en-us/articles/reducing-the-impact-of-misaligned-memory-accesses/

Optimization Notice

Compiler Intrinsics for Alignment

`__declspec(align(base, [offset]))`

Instructs the compiler to create the variable so that it is aligned on an “base”-byte boundary, with an “offset” (Default=0) in bytes from that boundary

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

Instructs the compiler to create a pointer to memory such that the pointer is aligned on an n-byte boundary

`#pragma vector aligned | unaligned`

Use aligned or unaligned loads and stores for vector accesses.

`__assume_aligned(a,n)`

Instructs the compiler to assume that array a is aligned on an n-byte boundary

"I've stopped using the Intel compiler. Each time I ship the product to a customer, they complain that applications crashes"!"

A games developer at a recent networking event.

Optimization Notice

Imagine this scenario:

1. Your IT dept have just bought you the latest and greatest Intel based workstation.

2. You've heard auto-vectorisation can make a real difference to performance

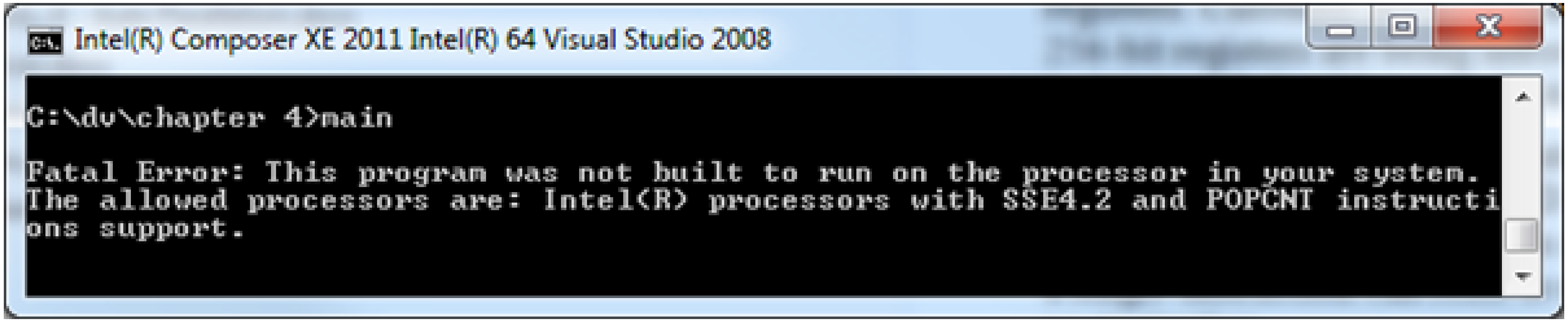
3. You enable auto-vectorisation using -xhost

4. You boast to your colleagues, "*my application runs faster than anything you can write…*"

5. You send the application to a colleague – it refuses to run.

What might be the issue? How can it be overcome?

Running a Mismatched Application

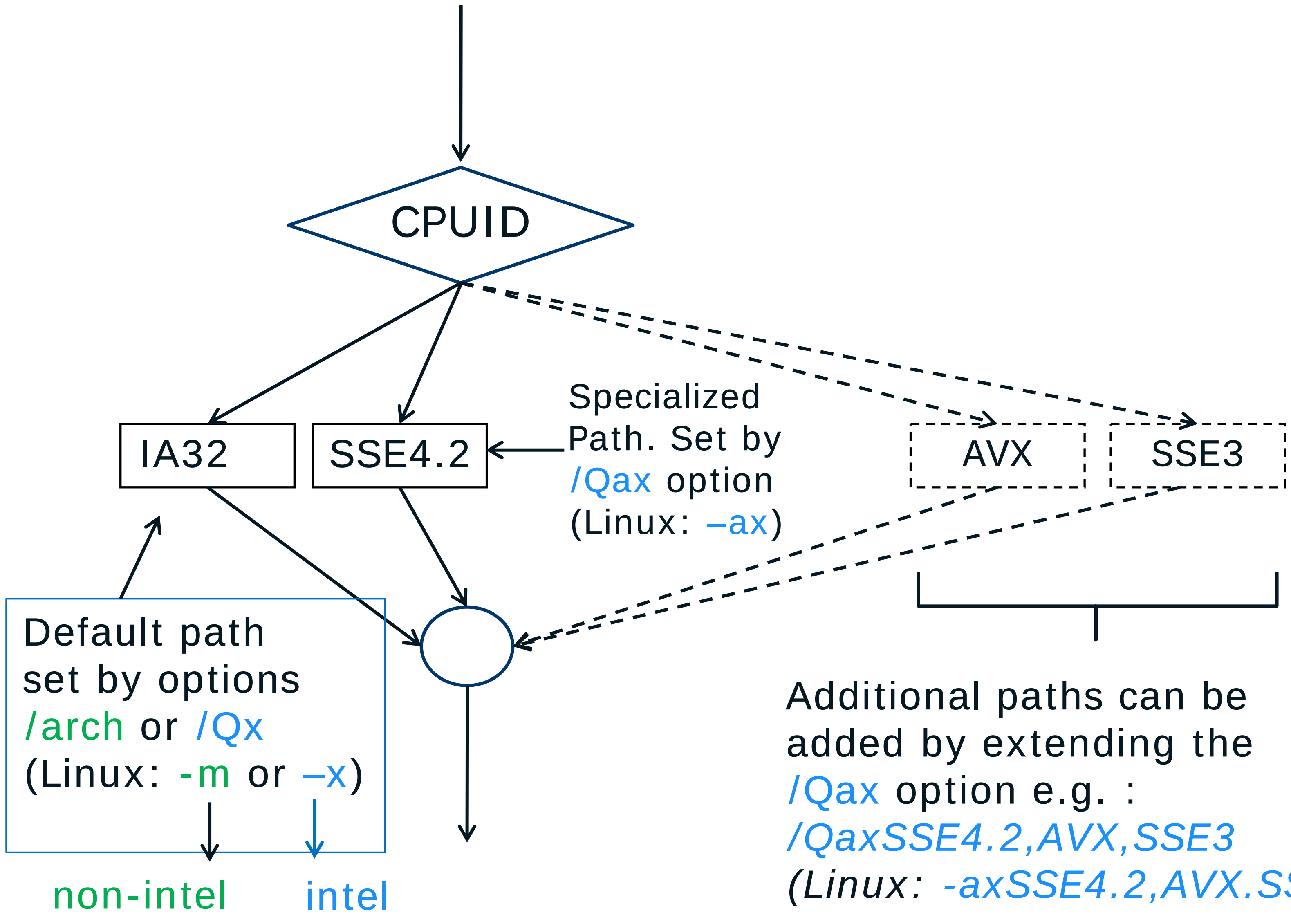


```
C:\dv\chapter 4>main

Fatal Error: This program was not built to run on the processor in your system.
The allowed processors are: Intel(R) processors with SSE4.2 and POPCNT instructi
ons support.
```

Step 3

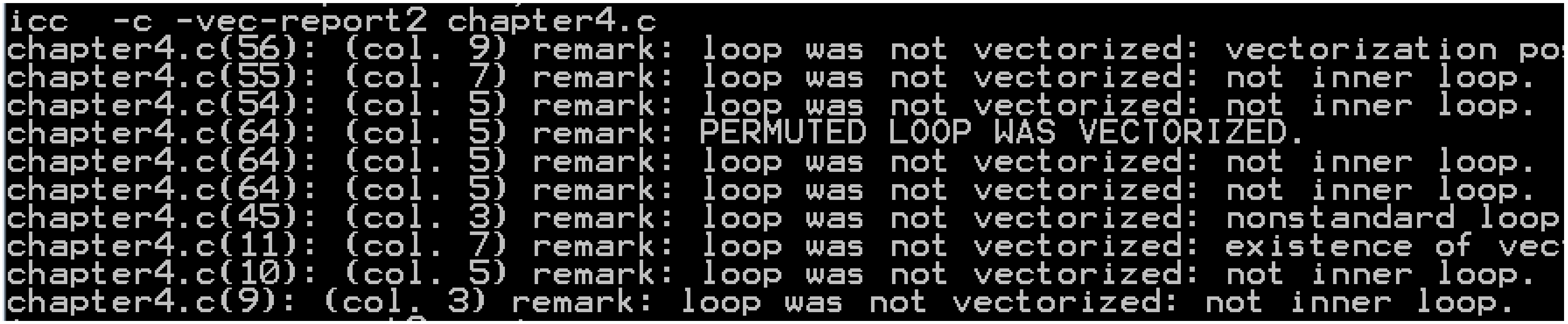
Multipath Auto-vectorisation


CPUID
IA32
SSE4.2
AVX
SSE3
Specialized Path. Set by /Qax option (Linux: –ax)
Default path set by options /arch or /Qx (Linux: -m or –x)
non-intel
intel

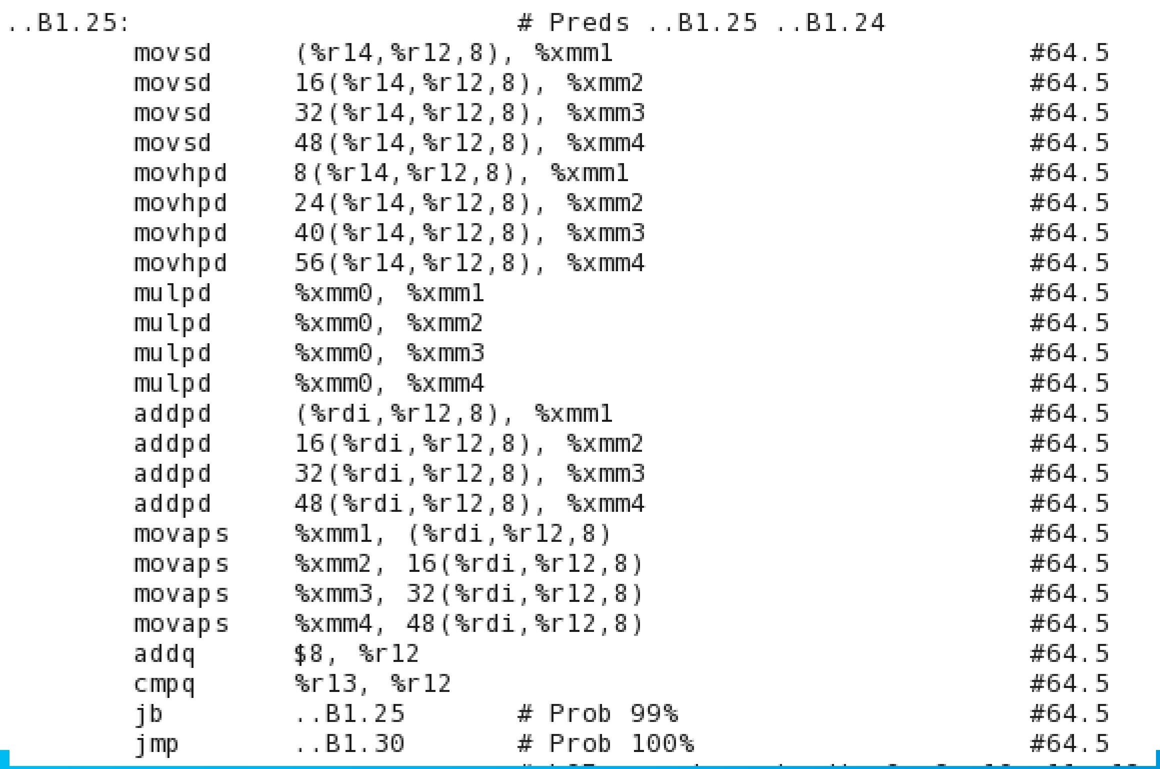
Additional paths can be added by extending the /Qax option e.g. : /QaxSSE4.2,AVX,SSE3 (Linux: -axSSE4.2,AVX.SSE3)

Step 3

The vectorised code uses Packed Instructions



```
icc  -c -vec-report2 chapter4.c
chapter4.c(56): (col. 9) remark: loop was not vectorized: vectorization po
chapter4.c(55): (col. 7) remark: loop was not vectorized: not inner loop.
chapter4.c(54): (col. 5) remark: loop was not vectorized: not inner loop.
chapter4.c(64): (col. 5) remark: PERMUTED LOOP WAS VECTORIZED.
chapter4.c(64): (col. 5) remark: loop was not vectorized: not inner loop.
chapter4.c(64): (col. 5) remark: loop was not vectorized: not inner loop.
chapter4.c(45): (col. 3) remark: loop was not vectorized: nonstandard loop
chapter4.c(11): (col. 7) remark: loop was not vectorized: existence of vec
chapter4.c(10): (col. 5) remark: loop was not vectorized: not inner loop.
chapter4.c(9): (col. 3) remark: loop was not vectorized: not inner loop.
```

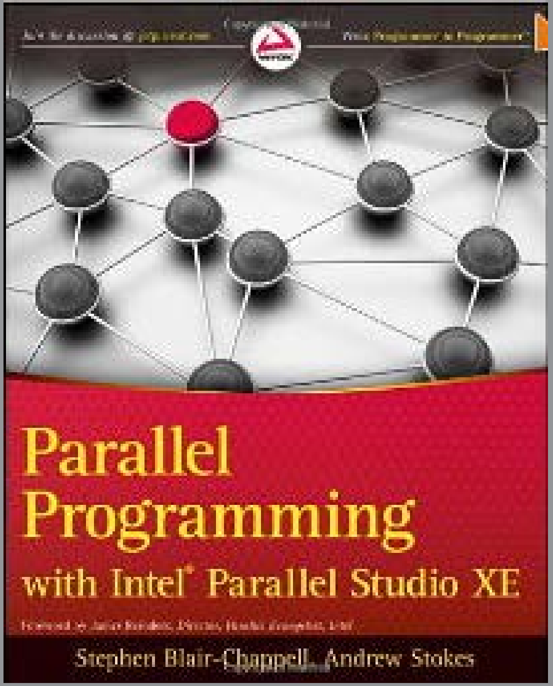


```
..B1.25:                        # Preds ..B1.25 ..B1.24
        movsd     (%r14,%r12,8), %xmm1                            #64.5
        movsd     16(%r14,%r12,8), %xmm2                          #64.5
        movsd     32(%r14,%r12,8), %xmm3                          #64.5
        movsd     48(%r14,%r12,8), %xmm4                          #64.5
        movhpd    8(%r14,%r12,8), %xmm1                           #64.5
        movhpd    24(%r14,%r12,8), %xmm2                          #64.5
        movhpd    40(%r14,%r12,8), %xmm3                          #64.5
        movhpd    56(%r14,%r12,8), %xmm4                          #64.5
        mulpd     %xmm0, %xmm1                                    #64.5
        mulpd     %xmm0, %xmm2                                    #64.5
        mulpd     %xmm0, %xmm3                                    #64.5
        mulpd     %xmm0, %xmm4                                    #64.5
        addpd     (%rdi,%r12,8), %xmm1                            #64.5
        addpd     16(%rdi,%r12,8), %xmm2                          #64.5
        addpd     32(%rdi,%r12,8), %xmm3                          #64.5
        addpd     48(%rdi,%r12,8), %xmm4                          #64.5
        movaps    %xmm1, (%rdi,%r12,8)                            #64.5
        movaps    %xmm2, 16(%rdi,%r12,8)                          #64.5
        movaps    %xmm3, 32(%rdi,%r12,8)                          #64.5
        movaps    %xmm4, 48(%rdi,%r12,8)                          #64.5
        addq      $8, %r12                                        #64.5
        cmpq      %r13, %r12                                      #64.5
        jb        ..B1.25       # Prob 99%                        #64.5
        jmp       ..B1.30       # Prob 100%                       #64.5
```

Step 3

Optimization Notice

Hands-on Lab


Parallel Programming with Intel® Parallel Studio XE
Stephen Blair-Chappell, Andrew Stokes

Activity 4-3

Using more Advanced Vectorisation

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

Review Sheet for Efficient Vectorization

- Are you using vector-friendly options such as –ansi-alias and –align array64byte?
- Are all hot loops vectorized and maximizing use of unit-stride accesses?
- Align the data and Tell the compiler
- Have you studied the vec-report6 output for hot-loops to ensure these?
- Are there any peel-loop and remainder-loop generated for your key-loops (Have you added loop_count pragma)?
  - Make changes to ensure significant runtime is not being spent in such loops
- Are you able to pad your arrays and get improved performance with –opt-assume-safe-padding?
- Have you added "#pragma vector aligned nontemporal" for all loops with streaming-store accesses to maximize performance?
- Avoid branchy code inside loops to improve vector-efficiency
  - Avoid duplicates between then and else, use builtin_expect to provide hint, move loop-invariant loads and stores under the branch to outside loops
- Use hardware supported operations only (rest will be emulated)

Review Sheet for Vectorization 2

- Use Intel Cilk Plus extensions for efficient and predictable vectorization
  - #pragma SIMD and !DEC$ SIMD
    - Counterpart of OMP for vectorization
  - Short-vector array notation for C/C++
    - Shifts burden to the user to express explicit vectorization
    - High-level and portable alternative to using intrinsics
  - Use elemental functions (C and Fortran) for loops with function calls
    - Can also be used to express outer-loop vectorization
- Study opportunities for outer-loop vectorization based on code access patterns
  - Use array-notations OR elemental-functions to express it
- Make memory accesses unit-strided in vector-loops as much as possible
  - Important for C and Fortran
- F90 array notation also can be used in short-vector form

Thank You

Legal Disclaimer & Optimization Notice

Optimization Notice