Producing Optimised Code

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
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
Often we are happy with out-of-the-box experience

When was the last time you looked at some documentation?
The Sample Application

- Initialises two matrices with a numeric sequence
- Does a Matrix Multiplication
The main loop (without timing & printf)

```c
// 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

```c
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 Optimization Steps

**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*

- `/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`)

*Linux*

- `/Qparallel` (`-parallel`)

---

*Other brands and names are the property of their respective owners.*

*Copyright © 2012, Intel Corporation. All rights reserved.*
The Seven Optimisation Steps

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

**Example options**

*Windows* (Linux)

- `/Od` (-O0)
- `/01,/02,/03` (-O1, -O2, -O3)
- `/Qsse4.2` (-xsse4.2)
- `/QxHOST` (-xhost)
- `/Qipo` (-ipo)
- `/Qprof-gen` (-prof-gen)
- `/Qprof-use` (-prof-use)
- `/Qguide` (-guide)

*Use Intel Family of Parallel Models*

- `/Qparallel` (-parallel)
Step 2

Using the General Optimisations
/O2 (-O2) **Optimize for maximum speed.**

- This option will create *faster code* in most cases.
- Optimizations include
  - scalar optimizations
  - inlining and some other
  - Inter-procedural optimizations between functions/subroutines in the same source file
  - **vectorization**
  - limited versions of a few other loop optimizations such as loop versioning and unrolling that facilitate vectorization.
This option is very similar to /O2 except that it omits optimizations that tend to increase object code size, such as the in-lining of functions. Generally useful where memory paging due to large code size is a problem, such as server and database applications.

Auto-vectorization is not turned on, even if it is invoked individually by its fine grained switch /Qvec. However, at /O1 the vectorization associated with array notations is enabled.
/O3 (-O3) optimizes for further speed increases.

- This includes all the /O2 optimizations, together with other High Level Optimizations.

- These high level optimizations include more aggressive strategies such as:
  - scalar replacement,
  - data pre-fetching,
  - loop optimization,
  - among others.
Loop Transformations

• Frequently (optimal) vectorization is possible only after adapting the loops before
• The compiler component responsible for these loop transformations is phase HLO – High Level Optimization
  – While HLO is active for optimization level O2 and O3, only O3 activates the full set of transformations and applies the transformations more aggressively
• Intel compilers provide detailed report on HLO activity:
  \{L&M\}: -opt-report -opt-report-phasehlo
  \{W\}: /Qopt-report /Qopt-report-phasehlo

... LOOP INTERCHANGE in loops at line: 7 8 9
Loopnest permutation ( 1 2 3 ) -->( 2 3 1 )

... Loop at line 7 unrolled and jammed by 4
Loop at line 8 unrolled and jammed by 4
...
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
Compiler Reports – 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)
Optimization Report Example

```
icc -O3 -opt-report-phase=hlo -opt-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
```
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_pspli, ilo, ilo_arg_prefetching, ilo_lowering, 
ilo_strength_reduction, ilo_reassociation, ilo_copy_propagation, 
ilo_convert_insertion, ilo_convert_remove, 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
```
Results of Running the Example Application

The graph shows the time (in seconds) for different general options across Core 2 Laptop, Xeon Server, SNB No TurboBoost, and SNB with TurboBoost. The x-axis represents the general options, while the y-axis represents the time in seconds.
An example of inlining

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

Source Code

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

Assemble Code

```
.B1.44:  # LUT
      movl  $.L2__STRING.0, %edi
      xorl  %eax, %eax
   .B1.41:  # Prob 100%
     jmp  .B1.41
   .B1.46:  # Infreq
      movl  $10, %edx
      movq  8(%r13), %rd1
     call  strto1
```

Step 2

Generate assembler file

Create a report
Step 3

Using Processor Specific Options
SIMD Instruction Enhancements

- **SSE** (1999):
  - 70 instr
  - Single-Precision Vectors
  - Streaming operations

- **SSE2** (2000):
  - 144 instr
  - Double-precision Vectors
  - 8/16/32 64/128-bit integer

- **SSE3** (2004):
  - 13 instr
  - Complex Data

- **SSSE3** (2006):
  - 32 instr
  - Decode

- **SSE4.1** (2007):
  - 47 instr
  - Video
  - Graphics building blocks
  - Advanced vector instr

- **SSE4.2** (2008):
  - 8 instr
  - String/XML processing
  - POP-Count
  - CRC

- **AES-NI** (2009):
  - 7 instr
  - Encryption and Decryption
  - Key Generation

- **AVX** (2010/11):
  - ~100 new instr.
  - ~300 legacy SSE instr
  - updated
  - 256-bit vector
  - 3 and 4-operand instructions

---

Step 3
Auto-Vectorization

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

```c
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.
Scalar and Packed Instructions

**add**<sub>SS</sub> Scalar Single-FP Add

**add**<sub>PS</sub> Packed Single-FP Add

---

**Single precision FP data**

**Scalar execution mode**

**Packed execution mode**

---

Step 3
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];
}

Examples of Code Generation

**SSE2**

```c
.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
```

**AVX**

```c
.B1.2::
  vmovaps  ymm3, A[rdx*8]
  vmovaps  ymm1, C[rdx*8]
  vcmpeqpd 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
```

**SSE4.1**

```c
.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
```

Step 3
## Results of Enhancing Auto-Vectorisation

<table>
<thead>
<tr>
<th>Setting</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE2</td>
<td>0.293</td>
<td>1</td>
</tr>
<tr>
<td>AVX</td>
<td>0.270</td>
<td>1.09</td>
</tr>
</tbody>
</table>
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”
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**
Stride Values for a 2 dimensional array

Programming conceptual model

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Physical storage in memory

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Consecutive row elements ‘stride’ distance is 4

Consecutive column elements ‘stride’ distance is 1
Overview of Writing Vector Code

Array Notation

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

Elemental Function

```c
__declspec(vector)
float ef(float a, float b) {
    return a + b;
}
A[::] = ef(B[::], C[::]);
```

SIMD Directive

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

Auto-Vectorization

```c
for (int i = 0; i < N; ++i) {
    A[i] = B[i] + C[i];
}
```
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 instructions support.
```
## Building for non-intel CPUs /arch: (-m)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>avx</td>
<td>AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2, and SSE.</td>
</tr>
<tr>
<td>sse4.2</td>
<td>SSE4.2 SSE4.1, SSSE3, SSE3, SSE2, and SSE.</td>
</tr>
<tr>
<td>sse4.1</td>
<td>SSE4.1, SSSE3, SSE3, SSE2, and SSE instructions.</td>
</tr>
<tr>
<td>ssse3</td>
<td>SSSE3, SSE3, SSE2, and SSE instructions.</td>
</tr>
<tr>
<td>sse2</td>
<td>May generate Intel® SSE2 and SSE instructions.</td>
</tr>
<tr>
<td>sse</td>
<td>This option has been deprecated; it is now the same as specifying ia32.</td>
</tr>
<tr>
<td>ia32</td>
<td>Generates x86/x87 generic code that is compatible with IA-32 architecture.</td>
</tr>
</tbody>
</table>

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.
### Building for Intel processors /Qx (-x)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE-AVX2</td>
<td>AVX2, AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2, and SSE instructions.</td>
</tr>
<tr>
<td>CORE-AVX-I</td>
<td>RDND instr, AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2, and SSE instructions.</td>
</tr>
<tr>
<td>AVX</td>
<td>AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2, and SSE instructions.</td>
</tr>
<tr>
<td>SSE4.2</td>
<td>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.</td>
</tr>
<tr>
<td>SSE4.1</td>
<td>SSE4 Vectorizing Compiler and Media Accelerator, SSSE3, SSE3, SSE2, and SSE. May optimize for Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture.</td>
</tr>
<tr>
<td>SSSE3_ATOM</td>
<td>MOVBE, (depending on -minstruction), SSSE3, SSE3, SSE2, and SSE. Optimizes for the Intel® Atom™ processor and Intel® Centrino® Atom™ Processor Technology</td>
</tr>
<tr>
<td>SSSE3</td>
<td>SSSE3, SSE3, SSE2, and SSE. Optimizes for the Intel® Core™ microarchitecture.</td>
</tr>
<tr>
<td>SSE3</td>
<td>SSE3, SSE2, and SSE. Optimizes for the enhanced Pentium® M processor microarchitecture and Intel NetBurst® microarchitecture.</td>
</tr>
<tr>
<td>SSE2</td>
<td>SSE2 and SSE. Optimizes for the Intel NetBurst® microarchitecture.</td>
</tr>
</tbody>
</table>
Multipath Auto-vectorisation

CPUID

IA32  SSE4.2

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
Hands-on Lab

Activity 4-3 (Page 106)
Using Auto-vectorisation
The vectorised code uses Packed Instructions

```
# Predecesor B1.25 # Predecesors 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
Step 4

Using Inter Procedural Optimisation

-and its effect on Vectorisation
Interprocedural Optimisation

Step 4

Source files:
- f1.c
- f2.c

Intermediate language (mock) objects:
- f1.obj
- f2.obj

Libraries:
- .lib

Executable:
- f.exe

Compiles:
- f1.c
- f2.c

Intermediate language (mock) objects are compiled into object files.

Libraries are linked with object files.

The object files and libraries are linked to create the executable.

Keywords:
- Interprocedural Optimisation
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
Impact of IPO on Auto-Vectorisation

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

<table>
<thead>
<tr>
<th>Platform</th>
<th>O2</th>
<th>IPO</th>
<th>QXHOST</th>
<th>Speedup O2 to IPO</th>
<th>Speedup O2 to QXHOST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 2 Laptop</td>
<td>0.474</td>
<td>0.272</td>
<td>0.266</td>
<td>1.74</td>
<td>1.78</td>
</tr>
<tr>
<td>SNB</td>
<td>0.293</td>
<td>0.181</td>
<td>0.171</td>
<td>1.62</td>
<td>1.71</td>
</tr>
<tr>
<td>SNB Turbo</td>
<td>0.211</td>
<td>0.132</td>
<td>0.124</td>
<td>1.60</td>
<td>1.70</td>
</tr>
<tr>
<td>Xeon workstation</td>
<td>0.239</td>
<td>0.211</td>
<td>0.209</td>
<td>1.13</td>
<td>1.14</td>
</tr>
</tbody>
</table>
Multiple messages on one 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.
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;
}

**addy.c**
double AddY( double sumy, int k )
{
    // sumy--; sumy = sumy + (double)k;
    return sumy;
}

This modification results in a 20% boost of performance
Hands-on Lab

Activity 4-4 (Page 111)
Using Interprocedural Optimisation (IPO)
Results so far ...

![Graph showing optimization steps and time (seconds)]

- **Optimization Steps**: novec, O2, AVX, IPO, Fix, PGO
- **Time (Seconds)**: 0 to 0.3
- **Lower is better**
- **Speedup**: $0.211 / 0.064 = 3.2$

*Step 5*
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

1. Application Source C/C++/Fortran
2. Compiler
3. Application Binary + Opt Reports
4. Performance Tools
5. Identify hotspots, problems

---

1. Application Source + Hotspots
2. Compiler in advice-mode
3. Advice messages
4. Modified Application Source
5. Compiler (extra options)
6. 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\}: \texttt{-guide-opts=<string>} \quad \{W\}: \texttt{/Qguide-opts:<string>}

Samples for \texttt{<string>}:
- “init.c, 1-50,100-150”
  \hspace{1cm} Restrict analysis to file init.c, lines 1-50 and 100-150
- “bar.f90,’m1::func_solve’”
  \hspace{1cm} Restrict analysis to file bar.f90, Fortran module “m1”,
  \hspace{1cm} function ‘func_solve’

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

\{L&M\}:
- \texttt{-guide-file=<file_name>}
- \texttt{-guide-file-append=<file_name>}

\{W\}:
- \texttt{/Qguide-file:<file_name>}
- \texttt{/Qguide-file-append:<file_name>
Vectorization Example [1]

```c
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]

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

```c
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];
    }
}
```
Data Transformation Example

```c
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
};
```

```c
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 ...
Additional Info

More on Auto-vectorisation
**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

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

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

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.

<table>
<thead>
<tr>
<th>3-Operand Example</th>
<th>4-Operand Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmm10 = xmm9 + xmm1</code></td>
<td><code>xmm10 = xmm9 + m128</code></td>
</tr>
<tr>
<td><code>movaps xmm10, xmm9</code></td>
<td><code>movups xmm10, m128</code></td>
</tr>
<tr>
<td><code>addpd xmm10, xmm1</code></td>
<td><code>vaddpd xmm10, xmm9, xmm9</code></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>4-Operand Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>movaps xmm0, xmm4</code></td>
</tr>
<tr>
<td><code>movaps xmm1, xmm2</code></td>
</tr>
<tr>
<td><code>vblendvps xmm1, xmm2, m128, xmm4</code></td>
</tr>
<tr>
<td><code>blendvps xmm1, m128</code></td>
</tr>
</tbody>
</table>
Intel® Microarchitecture (Sandy Bridge) Highlights

Instruction Fetch & Decode ➔ Allocate/Rename/Retire

Scheduler (Port names as used by IACA)
- Port 0
  - ALU
  - VI MUL
  - SSE MUL
  - DIV *
  - AVX FP MUL
- Port 1
  - ALU
  - VI ADD
  - SSE ADD
- Port 5
  - ALU
  - AVX/FP Shuf
  - Imm Blend
- Port 2
  - Load
  - JMP
  - AVX/FP Bool
  - Imm Blend
- Port 3
  - Load
  - Store Address
- Port 4
  - Store Address
  - STD

Memory Control
- 48 bytes/cycle

L1 Data Cache

- 1-per-cycle 256-bit multiply, add, and shuffle
- Load double the data with Intel microarchitecture (Sandy Bridge) !!!
## Other Ways of Inserting Vectorised Code

<table>
<thead>
<tr>
<th>Ease of use</th>
<th>Programmer control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Performance Libraries (e.g. IPP and MKL)</td>
<td>Manual CPU Dispatch (__declspec(cpu_dispatch ...))</td>
</tr>
<tr>
<td>Compiler: Fully automatic vectorization</td>
<td>SIMD intrinsic class (F32vec4 add)</td>
</tr>
<tr>
<td>Cilk Plus Array Notation</td>
<td>Vector intrinsic (mm_add_ps())</td>
</tr>
<tr>
<td>Compiler: Auto vectorization hints (#pragma ivdep, ...)</td>
<td>Assembler code (addps)</td>
</tr>
</tbody>
</table>

*Other brands and names are the property of their respective owners.*
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
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
0 1 2 3 4 5 6 7 8
The Section Operator (:)  

Array[ lower bound : length : stride].

```c
int C[5][5]
```

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

```plaintext
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
```
The Section Operator (,:)  

Array[ lower bound : length : stride].

D[ 0: 3: 2]
C/C++ Operators

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

\[ z[\cdot] = x[\cdot] \times y[\cdot] \quad \text{// element-wise multiplication} \]

\[ c[3:2][3:2] = a[3:2][3:2] + b[5:2][5:2] \quad \text{// 2x2 matrix addition} \]

Equivalent to

\[
\begin{align*}
&c[3][3] = a[3][3] + b[5][5]; \\
&c[3][4] = a[3][4] + b[5][6]; \\
&c[4][3] = a[4][3] + b[6][5]; \\
&c[4][4] = a[4][4] + b[6][6]; \\
\end{align*}
\]

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.

\[
a[b[0:s]] = c[:] \quad \Rightarrow \quad \text{for}(i=0; i<s; i++) \quad a[b[i]]=c[i];
\]

\[
c[0:s] = a[b[:]] \quad \Rightarrow \quad \text{for}(i=0; i<s; i++) \quad c[i]=a[b[i]];\]

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.

```c
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

```c
// 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]);
```
Function Maps

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

```c
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?

```c
#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][0] * b[0][j];
        }
    }
}
```

mm.c(9): error: rank mismatch in array section expression
```
c[i][j] += a[i][0] * b[0][j];
```
Every expression has a \textit{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 \textit{common rank} of two expressions is

\begin{itemize}
  \item if the rank of either expression is zero, the common rank is the rank of the other expression;
  \item otherwise, if the expressions have the same rank, that is the common rank;
  \item otherwise, the program is ill-formed.
\end{itemize}

(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 ($\texttt{postfix-expression } [ \texttt{section-triplet} ]$) 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 ($\texttt{postfix-expression } [ \texttt{expression} ]$) 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 \textit{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";


<table>
<thead>
<tr>
<th>Expression</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[3:4][0:10]</td>
<td>2</td>
</tr>
<tr>
<td>A[3][0:10]</td>
<td>1</td>
</tr>
<tr>
<td>A[3:4][0]</td>
<td>1</td>
</tr>
<tr>
<td>A[:][:]</td>
<td>2</td>
</tr>
<tr>
<td>A[3][0]</td>
<td>0</td>
</tr>
</tbody>
</table>
Square matrices

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

their matrix products are:

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

Source: http://en.wikipedia.org/wiki/Matrix_multiplication
for (i = 0; i < N; i++)
{
    for (j = 0; j < N; j++)
    {
        c[i][j] __sec_reduce_add = a[i][i] * b[i][j];
    }
}
Hands-on Lab

Activity 4-6 (New)
Cilk Plus array Notation
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

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

```c
#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\nand 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;
}
```
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.

Copyright © , Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and Cilk
are trademarks of Intel Corporation in the U.S. and other countries.

**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