The Software Vectorization Handbook

Applying Multimedia Extensions for Maximum Performance

Aart J. C. Bik

INTEL
PRESS

Copyright © 2004 Intel Corporation. All rights reserved.

ISBN 0-9743649-2-4

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4744. Requests to the Publisher for permission should be addressed to the Publisher, Intel Press, Intel Corporation, 2111 NE 25 Avenue, JF3-330, Hillsboro, OR 97124-5961. E-Mail: intelpress@intel.com

Intel and Pentium are registered trademarks of Intel Corporation.

† Other names and brands may be claimed as the property of others.

These pages were excerpted from Chapter 9 of *The Sofware Vectorization Handbook* by Aart Bik. Visit Intel Press to learn more about this book.

Vectorization Guidelines

To obtain high software performance, you must carefully select the proper algorithms and then implement them efficiently. Even though an expert programmer usually excels in both areas, the size of most applications renders hand optimizing all parts impossible. A more practical approach to improving software performance, therefore, is to let a compiler optimize the program as a whole and then to run a performance analysis to determine where additional optimization is required. This subsequent optimization may simply consist of rewriting the source into a form that is more amenable to optimization or, as a last resort, hand optimizing the code with intrinsics or inline assembly. This section provides some guidelines that may help the Intel C++ and Fortran compilers exploit multimedia extensions.

Design and Implementation Considerations

Decisions made during both the design and implementation phase of an application can ultimately have a profound impact on resulting performance. While optimizing an existing application, time or resource constraints usually prohibit reversing decisions that turn out to affect performance adversely. Therefore, selecting appropriate algorithms and data structures during the design and early implementation phase of an application is probably the most critical step toward high software performance.

Topics related to algorithm selection are clearly outside the scope of this book (Garey and Johnson 1999; Hopcroft and Ullman 1979; Knuth 1997a, 1997b, and 1997c; Sedgewick 1988). The only guideline that can be given in this context is that it may be worthwhile to take amenability to vectorization into consideration while deciding among several, equally suited algorithms. However, one algorithm should not be favored over another that has lower computational complexity, merely because the former is more amenable to vectorization. To take this argument to the extreme, given a choice between the following two ways to compute

$$\sum_{i=1}^{n} i$$

favoring the $O(n)$ function `fn()` over the $O(1)$ function `f1()` simply because the reduction can be vectorized seems rather senseless.

```
int fn(short n) {
  int r = 0, i;                              int f1(short n) {
  for (i = 1; i <= n; i++) {                   return (n > 0)
    r += i;                    which one?        ? (n*(n+1))/2
  }                                ↔            : 0;
  return r;                                    }
}
```

On the other hand, given the choice between two algorithms with identical computational complexity, secondary considerations such as being more amenable to vectorization may become a deciding factor.

Selecting appropriate data structures can also make vectorization of the resulting code more effective. To illustrate this point, compare the traditional array-of-structures (AoS) arrangement for storing the x_1, x_2, x_3, and x_4 components of a set of four-dimensional points with the alternative structure-of-arrays (SoA) arrangement for storing this set.

```
/* AoS */                  /* SoA */
struct {                   struct {
  double x1;                 double x1[N];
  double x2;       ↔         double x2[N];
  double x3;                 double x3[N];
  double x4;                 double x4[N];
} points[N];               } pointset;
```

With the AoS arrangement, a loop that visits all components of a point before moving to the next point exhibits a good locality of reference because all elements in fetched cache lines are utilized.

```
/* AoS */
for (i = 0; i < N; i++) {
  points[i].x1 *= scale1;
  points[i].x2 += trans2;
  points[i].x3 += trans3;
  points[i].x4 = 0;
}
```

The disadvantage of the AoS arrangement, however, is that each individual memory reference in such a loop exhibits a non-unit stride, which, in general, adversely affects vector performance. Furthermore, a loop that visits only one component of all points exhibits less satisfactory locality of reference because many of the elements in the fetched cache lines remain unused. In contrast, with the SoA arrangement, the loop shown above is expressed as follows:

```
/* SoA */
for (i = 0; i < N; i++) {
  pointset.x1[i] *= scale1;
  pointset.x2[i] += trans2;
  pointset.x3[i] += trans3;
  pointset.x4[i] = 0;
}
```

These unit-stride memory references are more amenable to effective vectorization and still exhibit good locality of reference within each of the four data streams. Note that cache anomalies that may reduce the memory performance of such separate streams can be avoided by applying loop distribution before vectorization. Additionally, loops that visit only one component of all points still exhibit the same favorable characteristics. Consequently, an application that uses the SoA arrangement may ultimately outperform an application based on the AoS arrangement when compiled with a vectorizing compiler, even if this performance difference is not directly apparent during the early implementation phase.

In other situations, the AoS arrangement may be the data structure of choice, as illustrated by the following operation, which adds all components of one point to another point:

```
/* AoS */
points[0].x1 += points[1].x1;
points[0].x2 += points[1].x2;
points[0].x3 += points[1].x3;
points[0].x4 += points[1].x4;
```

The loop materialization method presented in Chapter 8 maps this operation into only a few two-way SIMD parallel instructions. In contrast, the SoA arrangement now provides no opportunities for vectorization.

```
/* SoA */
pointset.x1[0] += pointsset.x1[1];
pointset.x2[0] += pointsset.x2[1];
pointset.x3[0] += pointsset.x3[1];
pointset.x4[0] += pointsset.x4[1];
```

This example illustrates the importance of selecting a data structure that effectively accommodates all frequently occurring operations. In some cases, hybrid data structures or even runtime conversions between two different formats may be required to expose the most available parallelism to a vectorizing compiler.

To illustrate another important consideration while selecting appropriate data structures, consider the following function to sum all elements in an array with data type `short`:

```
int add(short *a, int n) {
  int r = 0;
  while (--n >= 0) r += *a++;
  return r;
}
```

The Intel C/C++ Compiler is able to recognize a well-behaved loop in the `while` statement, which is subsequently vectorized, as can be seen in the following command line session on a source file `add.c`, which contains this function starting at line 10:

```
=> icl -Fa -QxP add.c
...
add.c(12) : (col. 3) remark: LOOP WAS VECTORIZED.
...
=> type add.asm
       ⋮
.B1.9:
```

```
       movq       xmm1, QWORD PTR [esi+ecx*2] ;12.26
       punpcklwd xmm1, xmm1                  ;12.26
       psrad      xmm1, 16                   ;12.26
       paddd      xmm0, xmm1                 ;12.20
       add        ecx, 4                     ;12.3
       cmp        ecx, eax                   ;12.3
       jb         .B1.9                      ;12.3
       ⋮
```

The accumulation of words into a doubleword accumulator is recognized as one of the idioms discussed in Chapter 7. The vector loop exploits four-way SIMD parallelism while computing partial sums, which are subsequently added into one final scalar value in a postlude. Now suppose that the programmer knows that either only the lower 16-bits of the sum are eventually used at all call sites, or the nature of all arrays passed to this function is such that the sum never exceeds a 16-bit precision. In that case, simply changing the data type of the accumulator into `short` enables the compiler to generate more efficient vector code, as follows:

```
=> icl -Fa -QxP add.c
...
add.c(12) : (col. 3) remark: LOOP WAS VECTORIZED.
...
=> type add.asm
       ⋮
.B1.9:
       paddw      xmm0, XMMWORD PTR [esi+ecx*2] ;12.20
       add        ecx, 8                        ;12.3
       cmp        ecx, eax                      ;12.3
       jb         .B1.9                         ;12.3
       ⋮
```

Inspecting the generated assembly file reveals that the vector loop now exploits eight-way SIMD parallelism while computing partial sums.

This example clearly shows that a relatively simple source code modification makes the difference between four-way or eight-way SIMD parallelism in the generated vector code. Because a compiler is usually not able to make such modifications, the data type of even a simple accumulator should be carefully selected to maximize the amount of available parallelism in the program. A general guideline in the context of intra-register vectorization, therefore, is to use the smallest data type possible for each data structure, since this ultimately enables the most parallelism in the resulting code.

Focus of Optimization

A generally useful optimization guideline is a result of Amdahl's law (Amdahl 1967), which states that if ρ denotes the fraction of dynamically executed instructions that inherently cannot exploit any parallelism in a program, then the maximum obtainable speedup S for this program with w-way parallelism is bounded by the following formula:

$$S \leq \frac{1}{\rho + \frac{(1-\rho)}{w}} \leq \frac{1}{\rho} (\text{for } w \to \infty)$$

As an example, if 80 percent of the execution time of a program is spent in inherently sequential code ($\rho = 0.8$), then the best speedup that can be obtained by vectorizing the remaining code is bounded by $S \leq 1.25$. This implies that, before optimizing a program with multimedia extensions, the maximum obtainable speedup should first be estimated because the program may reap little benefit from SIMD parallelism, no matter how much effort is put into the optimization.

If vectorization seems profitable, Amdahl's law also reveals what parts of a program benefit the most from using multimedia extensions. For example, suppose a program spends 2 percent of its execution time in initialization code and the remaining 98 percent in a computational kernel, and both code regions can be vectorized. If time constraints allow for the optimization of *only one* code region ($\rho_1 = 0.98$ and $\rho_2 = 0.02$), then clearly the most benefits can be expected from vectorizing the kernel ($S_1 \leq 1.02$ vs. $S_2 \leq 50$). As a result, when optimizing a program, the first focus should be on the hot spots—the parts of the program with intense activity (Gerber 2002). A software profiler like the Intel VTune™ Performance Analyzer (Intel 1998–2004b) is an essential tool to detect such regions. Due to diminishing returns for less frequently executed regions of a program, these colder regions should be optimized only after all other means of improvement have been exhausted.

Diagnostics-Guided Optimization

Vectorization diagnostics are useful to determine where automatic vectorization has been successful and where it has failed. Based on this information, the programmer can apply code restructuring or insert compiler hints to make important hot spots that remain sequential more amenable to vectorization or to disable the vectorization of every code fragment that adversely affects performance.

Table 9.5 shows common vectorization diagnostics. Compiler switch `-Qvec_report1` provides feedback on successful vectorization of loops, distributed loops, and straightline code fragments. In addition, switch `-Qvec_report2` provides feedback on loops that fail to vectorize, followed by a short description of the reason for failure. The table shows only a few examples. Note that, as compiler capabilities improve, later versions of the compiler may provide more detailed failure descriptions or even report success where former versions reported failure. The message `vectorization possible but seems inefficient`, for example, indicates that the efficiency heuristics of the compiler deem vectorization unprofitable, even though vector code could potentially be generated. As explained earlier, this decision is simply overridden with a compiler hint `#pragma vector always` before the loop. Not all vectorization failures are easily remedied, however. Data dependences, for example, may reflect execution order constraints that prevent vectorization altogether. In such cases, `-Qvec_report3` may provide more insights on whether any of the data dependence eliminating techniques described in the literature (Padua and Wolfe 1986, Polychronopoulos 1988, Wolfe 1996, Zima and Chapman 1990) may help to enable vectorization.

Table 9.5 Common Vectorization Diagnostics

Windows Switch	Diagnostic
-Qvec_report1 (and higher)	LOOP WAS VECTORIZED PARTIAL LOOP WAS VECTORIZED BLOCK LOOP WAS VECTORIZED
-Qvec_report2 (and higher)	loop was not vectorized: existence of vector dependence low trip count mixed data types not inner loop operator unsuited for vectorization subscript too complex statement cannot be vectorized unsupported loop structure vectorization possible but seems inefficient #pragma novector used ⋮
-Qvec_report2	vector dependence: proven [FLOW/ANTI/OUTPUT] dependence between ... assumed [FLOW/ANTI/OUTPUT] dependence between ...

To illustrate diagnostics-guided optimization, suppose that most of the execution time of an application is spent in the following function, which computes the product of a matrix and a vector:

```
int size1, size2;

void matvec(double a[][N], double b[], double x[]) {
  int i, j;
  for (i = 0; i < size1; i++) {       /* line 14 */
    b[i] = 0;
    for (j = 0; j < size2; j++) {     /* line 16 */
      b[i] += a[i][j] * x[j];
    }
  }
}
```

If this function is defined at line 12 in source file `ax.c`, the following vectorization diagnostics result when the program is compiled on the command line as follows:

```
=> icl -Fa -QxN -Qvec_report2 ax.c
...
ax.c(14) : (col. 3) remark: loop was not vectorized:
                            not inner loop.
ax.c(16) : (col. 5) remark: loop was not vectorized:
                            unsupported loop structure.
...
```

Although failure to vectorize the outer loop is expected, the diagnostic on an unsupported loop structure at line 16 comes somewhat as a surprise. This complication arises from using a global variable `size2` as upper bound, because type-unaware aliasing assumptions prevent the compiler from proving loop invariance of this variable and, hence, detecting a well-behaved inner loop. Even though interprocedural aliasing analysis—enabled with either `-Qip` or `-Qipo`—eliminates most of such false aliasing assumptions, this particular complication is also easily remedied while compiling the function in isolation by means of the following source code modification:

```
void matvec(double a[][N], double b[], double x[]) {
  int i, j, sz2 = size2;
  for (i = 0; i < size1; i++) {    /* line 14 */
    b[i] = 0;
    for (j = 0; j < sz2; j++) {    /* line 16 */
      b[i] += a[i][j] * x[j];
    }
  }
}
```

Vectorization diagnostics now report successful vectorization:

```
=> icl -Fa -QxN -Qvec_report2 ax.c
...
ax.c(14) : (col. 3) remark: loop was not vectorized:
                    not inner loop.
ax.c(16) : (col. 5) remark: LOOP WAS VECTORIZED.
...
```

At this point, programmers can typically start focusing on the next hot spot in the program. Inspection of the corresponding assembly file, however, would reveal that the compiler resorted to dynamic data dependence testing and loop peeling to handle the possibility of overlap between the left-hand and right-hand side expressions and misalignment, respectively. Although these dynamic methods provide a convenient way to improve performance of the function when no further information is available, in some cases the associated runtime overhead can be avoided altogether. Suppose, for example, that a programmer with some knowledge of the application domain realizes that this function is exclusively applied to disjoint memory regions that are 16-byte aligned. This additional information is easily conveyed to the compiler by means of the following compiler hints:

```
void matvec(double a[][N], double b[], double x[]) {
  int i, j, loc_size = size;
__assume_aligned(a, 16);
__assume_aligned(b, 16);
__assume_aligned(x, 16);
  for (i = 0; i < loc_size; i++) {
    b[i] = 0;
#pragma ivdep
    for (j = 0; j < loc_size; j++) {
      b[i] += a[i][j] * x[j];
    }
  }
}
```

The alignment pragmas together with the statically known value of `N` enable the compiler to determine the alignment of all access patterns in the innermost loop. The other pragma enables the compiler to discard all assumed data dependences. Figure 9.1 shows performance results on a 3-gigahertz Pentium 4 processor for varying matrix sizes. These results are based on execution times that were obtained by running the function `matvec()` repeatedly and dividing the total runtime accordingly. Speedups are shown for vector versions that use the default dynamic optimizations (`S-vec`), the alignment pragmas (`S-align`), the data dependence

pragma (S-ivdep), or both sets of pragmas (S-both). These results show that, although the performance improvements obtained with fully automatic vectorization are already quite satisfactory—speedups in the range 1.4 to 2.7 for the given matrix sizes, compiler hints can help to squeeze more performance out of this fragment—speedups in the range 2.3 to 3.1. For an application that spends most of its execution time computing the product of a matrix and a vector, this extra improvement can be quite noticeable in the performance of the application as a whole.

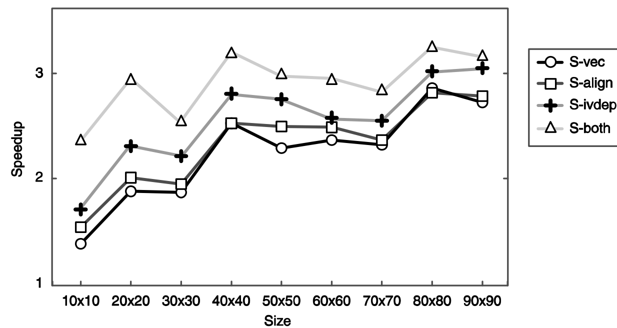

Figure 9.1 Matrix x Vector Speedup—3-Gigahertz Pentium® 4 Processor

Fully automatic vectorization, possibly assisted by minor source code modifications and compiler hints, is clearly the preferred way to exploit multimedia extensions. The approach preserves portability of the program, simplifies maintenance, and allows future releases of the vectorizing compiler to take advantage of the latest multimedia extensions. Despite ongoing efforts to improve compiler technology, however, hand optimizing a fragment with intrinsics or inline assembly is sometimes the only viable way to exploit multimedia extensions. Even with this last resort, preserving the original source code by means of a macro mechanism similar to the one following is highly recommended:

```
#ifdef __HANDOPT
   /* hand optimized implementation */
   _asm {
     ...
   }
#else
   /* original source code */
   ...
#endif
```

The original source code typically reflects the functionality of the code fragment more clearly, which simplifies maintenance and even enables an occasional comparison of the quality of the hand-optimized implementation with code that is generated by different or newly released compilers. The hand-optimized implementation is simply integrated into the executable by compiling the program with the additional switch `-D__HANDOPT` to define the macro name. For a detailed description of intrinsics and inline assembly, refer to the Intel documentation (Intel 1998-2004a, 2003a, 2003b, 2003c, 2003d).