VISUAL STUDIO 2012 : AUTO-VECTORIZATION COOKBOOK

INTRODUCTION

Auto-vectorization is a feature in the C++ Compiler for Visual Studio 2012. It analyzes loops in your C++ programs and tries to make them run faster by using SSE instructions and the XMM vector registers, present in all modern PC processors.

The feature has shipped in Preview and Release-Candidate versions of Visual Studio 2012. For more information, please see the following blog links:

- Short overview
- Longer discussion
- Series of blog postings

This document provides a "cookbook" of tips and tricks for writing your C++ loops in ways that help the auto-vectorizer improve the performance of your code.

Note that these rules apply to version 1 of the auto-vectorizer. Some, such as LOOP-BODY- 6 (see below), are fundamental, and will remain. Others, such as LOOP BODY 8 are less so – we will work to remove many of these constraints in future releases, so expect to see this cookbook shrink in size.

Recall that the compiler provides two diagnostic switches:

- /Qvec-report:1 will report each loop that is successfully vectorized
- /Qvec-report:2 will, in addition, report each loop that failed to vectorize, with a corresponding "reason code".

RULES GENERAL

1. Compile your code, optimized for speed, */O2*, or */O2 /GL*, to ensure the auto-vectorizer kicks in
2. If writing tests, ensure the results calculated by a loop are actually used – else the compiler may optimize away the loop body, and report "not vectorized"

RULES FOR LOOP HEADER

1. Use *for* loops, rather than *while* loops.
2. Make your *for* loops count up in steps of +1.
3. Use type *int* or *size_t* for the loop counter.
4. The loop counter must be local (to the loop, or the function) – not global.
5. Increment the loop counter as the final statement in the *for* header – and nowhere else.
6. The lower bound can be any expression.
7. The upper bound must evaluate to a number whose value is constant for the duration of the loop.
8. A loop with small trip-count may not be vectorized. (The compiler will decide automatically whether to apply other, more effective, optimizations).

RULES FOR LOOP BODY

1. Only the innermost loop in a nest is a candidate for auto-vectorization.
2. Keep the loop body simple: no function calls, no if, no *break*, no continue, no *? :* and no *switch*. No exception handling.
3. It is ok, however, to call any of the intrinsic functions in *<math.h>* within the loop body.
4. Certain *if* patterns – implementing a *min* or *max* idiom – are ok.
5. A loop with that does little work (eg: *a[n] = 3*) won't be vectorized.
6. Avoid loops where any iteration uses the result of a previous iteration.
7. Avoid datatype conversions, explicit or implicit.
8. Ensure any bit-shift operations specify a shift amount that is constant for the loop.
9. Reductions (eg: calculating the sum or average of an array) require you to specify the */fp:fast* option for datatype float or double.
10. Use array types of *int*, *float* or *double*. Avoid *char* or *short* (*signed* or *unsigned*).
11. Do not mix operations on arrays of different element type in the same loop.
12. Do not use the value of scalar (non-array) variables beyond the loop.
13. If a loop accesses fields within a *struct*, those fields must be 32 or 64 bits wide.
14. Except for *struct* accesses, avoid accesses to memory that are not contiguous between successive iterations.
15. Auto-vectorization is incompatible with the compiler options: */kernel*, */arch:IA32*, */favor:ATOM*, */O1* or */Os*

REASON CODES

This section lists the reason codes that the auto-vectorizer emits, under */Qvec-report:2*

Reason Code	Explanation
500	This is a generic message – it covers several cases: for example, the loop includes multiple exits, or the loop header does not end by incrementing the induction variable
501	Induction variable is not local; or upper bound is not loop-invariant
502	Induction variable is stepped in some manner other than a simple +1
503	Loop includes Exception-Handling or switch statements
504	Loop body may throw an exception, requiring destruction of a C++ object

Reason Code	Explanation
1100	Loop contains control flow – if, ?:
1101	Loop contains datatype conversion, perhaps implicit, that cannot be vectorized
1102	Loop contains non-arithmetic, or other non-vectorizable operations
1103	Loop body includes shift operations whose size might vary within the loop
1104	Loop body includes scalar variables
1105	Loop includes a non-recognized reduction operation
1106	Inner loop already vectorized: cannot also vectorize outer loop

Reason Code	Explanation
1200	Loop contains loop-carried data dependences
1201	Array base changes during the loop
1202	Field within a struct is not 32 or 64 bits wide
1203	Loop body includes non-contiguous accesses into an array

Reason code 1200 says that the loop contains loop-carried data dependences which prevent vectorization. This means that different iterations of the loop *interfere* with each other in such a way that vectorizing the loop would produce wrong answers. More precisely, the auto-vectorizer cannot prove to itself that there are no such data-dependences.

Reason Code	Explanation
1300	Loop body contains no (or very little) computation
1301	Loop stride is not +1
1302	Loop is a “do-while”
1303	Too few loop iterations for vectorization to be a win
1304	Loop includes assignments that are of different size
1305	Not enough type information

Reason Code	Explanation
1400	User specified #pragma loop(no_vector)
1401	/kernel switch specified
1402	/arch:IA32 switch specified

1403	/arch:ATOM switch specified and loop includes operations on doubles
1404	/O1 or /Os switch specified

The 1400 block of reason codes is straightforward – the user specified some option that is just plain incompatible with vectorization.

Reason Code	Explanation
1500	Possible aliasing on multi-dimensional arrays
1501	Possible aliasing on arrays-of-structs
1502	Possible aliasing and array index is other than n + K
1503	Possible aliasing and array index has multiple offsets
1504	Possible aliasing – would require too many runtime checks
1505	Possible aliasing – but runtime checks are too complex

The block of 1500 reason codes is all about aliasing – where a location in memory can be accessed by two different names.

DETAILED EXAMPLES

This section explores each of the reason codes, listed in the previous section, providing examples and more details.

```
void code_500(int *A)
{
    // Code 500 is emitted if the loop has non-vectorizable flow.
    // This can include "if", "break", "continue", the conditional
    // operator "?", or function calls.
    // It also encompasses proper definition and use of the induction
    // variable "i", in that the increment "i++" must be the last statement
    // in the loop.

    int i = 0;
    while (i<1000)
    {
        if (i == 4)
        {
            break;
        }

        i++;

        A[i] = A[i] + 1;
    }

    // To resolve code 500, use a 'for' loop with single increment of
    // induction variable.

    for (int i=0; i<1000; i++)
    {
        A[i] = A[i] + 1;
    }
}
```

```
int bound();
void code_501_example1(int *A)
{
    // Code 501 is emitted if the compiler cannot discern the
    // induction variable of this loop. In this case, when checking
    // the upperbound of 'i', the compiler cannot prove that the
    // function call "bound()" returns the same value each time.
    // Also, the compiler cannot prove that the call to "bound()"
    // does not modify the values of array A.

    for (int i=0; i<bound(); i++)
    {
        A[i] = A[i] + 1;
    }

    // To resolve code 501, ensure the induction variable is
    // a local variable, and ensure the upperbound is a
```

```
    // provably loop invariant value.

    for (int i=0, imax = bound(); i<imax; i++)
    {
        A[i] = A[i] + 1;
    }
}
```

```
int i;
void code_501_example2(int *A)
{
    // Code 501 is emitted if the compiler cannot discern the
    // induction variable of this loop.  (One example is when
    // ‘i’ is global, as in this case:

    for (i=0; i<1000; i++)
    {
        A[i] = A[i] + 1;
    }

    // To resolve code 501, ensure the induction variable is
    // a local variable, and ensure the upperbound is a
    // provably loop invariant value.

    for (int i=0; i<1000; i++)
    {
        A[i] = A[i] + 1;
    }
}
```

```
void code_502(int *A)
{
    // Code 502 is emitted if the compiler cannot discern
    // the induction variable of the loop. In this case,
    // there are three increments to "i", one of which
    // is conditional.

    for (int i=0; i<1000; i++)
    {
        A[i] = A[i] + 1;
        i++;

        if (i < 100)
        {
            i++;
        }
    }

    // To resolve code 502, ensure that there is one single
    // increment of the induction variable, placed in the usual
    // spot in the "for" loop.

    for (int i=0; i<1000; i++)
    {
        A[i] = A[i] + 1;
    }
}
```

```
// compile with /EHsc
void code_503(int *A, int x)
{
    // Code 503 is emitted if there are inadmissible
    // operations in the loop. Exception handling and
    // switch statements, for example.

    for (int i=0; i<1000; i++)
    {
        try
        {
            A[i] = A[i] + 1;
        }
        catch (...)
        {

        }

        switch (x)
        {
            case 1: A[i] = A[i] + 1;
            case 2: A[i] = A[i] + 2;
            case 3: A[i] = A[i] + 3;
            break;
        }
    }

    // To resolve code 503, try to remove as many switch statements
    // and exception handling constructs as possible.
}
```

```
// compile with /EHsc

int might_throw_Cpp_exception();
class C504
{
public:
      C504();
      ~C504();
};

void code_504() {
    // Code 504 is emitted if a C++ object was created
    // that requires EH unwind tracking information under
    // /EHs or /EHsc.  In this example, the function
    // might_throw_Cpp_exception might throw a C++ exception,
    // which would then require the object ‘c’ be destroyed
    // as part of EH handling.

    for(int i = 0; i < 1000; ++i)
    {
        C504 c;
        A[i] = might_throw_Cpp_exception();
```

```
    }

}
void code_1100(int *A, int x)
{
    // Code 1100 is emitted when the compiler detects control flow
    // inside the loop. "if", the ternary operator "?", etc. Resolve
    // this by flattening or removing control flow inside the loop body.

    for (int i=0; i<1000; i++)
    {
        // straightline code is more amenable to vectorization
        if (x)
        {
            A[i] = A[i] + 1;
        }
    }
}
```

```
void code_1101(int *A, char *B)
{
    // Code 1101 is emitted when the compiler is unable to vectorize
    // convert operations in the loop body. Be mindful of C/C++ converts,
    // many of which are implicit.

    // In this example, the 1-byte load of "B[i]" is converted to 4-bytes
    // prior to the addition by "1". The VS11 compiler, although able to
    // vectorize most converts, does not vectorize char -> int converts.

    // Resolve this by eliminating conversions where possible.

    for (int i=0; i<1000; i++)
    {
        A[i] = B[i] + 1;
    }
}
```

```
extern "C"  long _InterlockedExchange(long * Target, long Value);

void code_1102(int *A, long *x)
{
    // Code 1102 is emitted when the compiler is unable to vectorize
    // an operation in the loop body. For example, some intrinsics and other
    // non-arithmetic, non-bit-manipulation, and non-memory operations are not
    // vectorizable.

    // Resolve this by removing as many non-vectorizable operations
    // as possible from the loop body.

    for (int i=0; i<1000; i++)
    {
        A[i] = A[i] + _InterlockedExchange(x, 1);
    }
}
```

```
void code_1103(int *A, int *B)
{
    // Code 1103 is emitted when the compiler is unable to vectorize
    // a "shift" operation. In this example, there are two shifts
    // that cannot be vectorized.

    for (int i=0; i<1000; i++)
    {
        A[i] = A[i] >> B[i]; // not vectorizable

        int x = B[i];
        A[i] = A[i] >> x;    // not vectorizable
    }

    // To resolve this, ensure that your shift amounts are loop
    // invariant. If it is not possible for shift amounts to be
    // loop invariant, it may not be possible to vectorize this loop.

    int x = B[0];
    for (int i=0; i<1000; i++)
    {
        A[i] = A[i] >> x; // vectorizable
    }
}
```

```
int code_1104(int *A, int *B)
{
    // When vectorizing a loop, the compiler must 'expand' scalar
    // variables to a vector size such that they can fit in
    // vector registers. Code 1104 is emitted when the compiler
    // cannot 'expand' such scalars.

    // In this example, we try to 'expand' x to be used in the
    // vectorized loop. Here, however, there is a use of 'x'
    // beyond the loop body, prohibiting this expansion.

    // To resolve this, try to limit your scalars to be used
    // only in the loop body (not beyond), and to keep their types
    // consistent with the loop types.

    int x;
    for (int i=0; i<1000; i++)
    {
        x = B[i];
        A[i] = A[i] + x;
    }

    return x;
}
```

```
int code_1105(int *A)
{
    // One operation that the compiler attempts to vectorize is called
    //  "reduction".  This is when operating on each element of an
    // array and computing a resulting scalar value, like this
    // piece of code, which computes the sum of each element in the array:

    int s = 0;
    for (int i=0; i<1000; i++)
    {
        s += A[i]; // vectorizable
    }

    // The reduction pattern must be similar to the above loop. The
    // compiler emits code 1105 if it cannot deduce the reduction
    // pattern. For example:

    for (int i=0; i<1000; i++)
    {
        s += A[i] + s;  // code 1105
    }

    // In the case of reductions over "float" or "double" types,
    // vectorization requires that the /fp:fast switch is thrown.
    // This is because vectorizing the reduction operation depends
    // upon "floating point reassociation".  Reassociation is only
    // allowed when /fp:fast is thrown.

    return s;
}
```

```
void code_1106(int *A)
{
    // Code 1106 is emitted when the compiler tries to vectorize
    // an outer loop.

    for (int i=0; i<1000; i++) // this loop is not vectorized
    {
        for (int j=0; j<1000; j++) // this loop is vectorized
        {
            A[j] = A[j] + 1;
        }
    }
}
```

```
void code_1200(int *A)
{
    // Code 1200 is emitted when data dependence is prohibiting
    // vectorization. This can only be resolved by rewriting your
    // loop.

    for (int i=0; i<1000; i++)
    {
        A[i] = A[i-1] + 1; // vectorization prohibiting
    }
}
```

```
void code_1201(int *A)
{
    // Code 1201 is emitted when an array base changes
    // in the loop body. Resolve this by rewriting your
    // code such that varying the array base is not necessary.

    for (int i=0; i<1000; i++)
    {
        A[i] = A[i] + 1;
        A++;
    }
}
```

```
typedef struct S_1202
{
    short a;
    short b;
} S_1202;

void code_1202(S_1202 *s)
{
    // Code 1202 is emitted when the loop body accesses struct fields
    // that are not 32 or 64 bits wide.

    for (int i=0; i<1000; i++)
    {
        s[i].a = s[i].b + 1; // this 16 bit struct access is not vectorizable
    }
}
```

```
void code_1203(int *A)
{
    // Code 1203 is emitted when the loop body includes complex memory
    // references.  (Note, in contrast, that VS11 supports vectorizing
    // some non-contiguous memory access, that require scatter/gather
    // accesses).

    for (int i=0; i<1000; i++)
    {
        A[i] += A[0] + 1;       // constant memory access not vectorized
        A[i] += A[i*2+2] + 2;  // non-contiguous memory access not vectorized
    }
}
```

```
void code_1300(int *A, int *B)
{
    // Code 1300 is emitted when the compiler detects no computation
    // in the loop body.

    for (int i=0; i<1000; i++)
    {
        A[i] = B[i]; // Do not vectorize, instead emit memcpy
    }
}
```

```
void code_1301(int *A)
{
    // Code 1301 is emitted when the stride of a loop is not positive 1.
    // VS11 only vectorizes loops with a stride of positive 1, and rewriting
    // your loop may be required.

    for (int i=0; i<1000; i += 2)
    {
        A[i] = A[i] + 1;
    }
}
```

```
void code_1302(int *A)
{
    // Code 1302 is emitted for "do-while" loops. VS11 only vectorizes
    // "while" and "for" loops.

    int i = 0;
    do
    {
        A[i] = A[i] + 1;
    } while (++i < 1000);
}
```

```
int code_1303(int *A, int *B)
{
    // Code 1303 is emitted when the compiler detects that
    // the number of iterations of the loop is too small to
    // make vectorization profitable.

    // Note that if the loop computation fits perfectly in
    // vector registers (eg. the upperbound is 4, or 8 in
    // this case) then the loop _may_ be vectorized.

    // This loop is not vectorized as there are 5 iterations: too few
    // to be profitable, and not an exact fit in the SSE registers.

    for (int i=0; i<5; i++)
    {
        A[i] = A[i] + 1;
    }

    // This loop is vectorized: although the trip count is small,
    // the calculation fits into one use of an SSE register.

    for (int i=0; i<4; i++)
    {
        A[i] = A[i] + 1;
    }

    // This loop is not vectorized because runtime pointer checks
    // are required to check that A and B don't overlap. It is not
    // worth it to vectorize this loop.

    for (int i=0; i<4; i++)
    {
        A[i] = B[i] + 1;
    }

    // This loop is not vectorized: although it fits into the SSE
    // vector registers, the extra scalar additions required, after
    // the vectorized loop, make it overall, unprofitable.

    int s = 0;
    for (int i=0; i<4; i++)
    {
        s += A[i];
    }
    return s;
}
```

```
void code_1304(int *A, short *B)
{
    // Code 1304 is emitted when the compiler detects statements in
    // the loop body whose result sizes differ.  In this case, the first
    // statement has a result size of 32 bits; the second, one of 16 bits.

    // Consider splitting the loop into loops to maximize
    // vector register utilization.

    for (int i=0; i<1000; i++)
    {
        A[i] = A[i] + 1;         // 32-bit result
        B[i] = B[i] + 1;         // 16-bit result
    }
}
```

```
typedef struct S_1305
{
    int a;
    int b;
} S_1305;

void code_1305( S_1305 *s, S_1305 x)
{
    // Code 1305 is emitted when the compiler can't discern
    // the type information required to vectorize a loop.
    // This includes struct assignments, as follows:

    // Resolve this by ensuring your loops have statements
    // which operate on integers or floating point types.

    for (int i=0; i<1000; i++)
    {
        s[i] = x;
    }
}
```

```
void code_1400(int *A)
{
    // Code 1400 is emitted when you specify the
    // no_vector pragma.

    #pragma loop(no_vector)
    for (int i=0; i<1000; i++)
    {
        A[i] = A[i] + 1;
    }
}
```

```
// Compile with /kernel
void code_1401(int *A)
{
    // Code 1401 is emitted when /kernel is specified.

    for (int i=0; i<1000; i++)
    {
        A[i] = A[i] + 1;
    }
}
```

```
// Compile with /arch:IA32
void code_1402(int *A)
{
    // Code 1401 is emitted when /arch:IA32 is specified.

    for (int i=0; i<1000; i++)
    {
        A[i] = A[i] + 1;
    }
}
```

```
// Compile with /favor:ATOM
void code_1403(double *A)
{
    // Code 1401 is emitted when /arch:ATOM is specified, and
    // the loop contains operations on "double" arrays.

    for (int i=0; i<1000; i++)
    {
        A[i] = A[i] + 1;
    }
}
```

```
// Compile with /O1 or /Os
void code_1404(int *A)
{
    // Code 1401 is emitted when compiling for size.

    for (int i=0; i<1000; i++)
    {
        A[i] = A[i] + 1;
    }
}
```

```
void code_1500(int A[100][100], int B[100][100])
{
    // Code 1500 is emitted when runtime pointer
    // disambiguation checks are required, and
    // there are multidimensional array references.

    for (int i=0; i<100; i++)
    {
        for (int j=0; j<100; j++)
        {
            A[i][j] = B[i][j] + 1;
        }
    }
}
```

```
typedef struct S_1501
{
    int a;
    int b;
} S_1501;

void code_1501(S_1501 *s1, S_1501 *s2)
{
    // Code 1501 is emitted when runtime pointer
    // disambiguation checks are required, and
    // there are array-of-struct accesses in the
    // loop body.

    for (int i=0; i<100; i++)
    {
        s1[i].a = s2[i].b + 1;
    }
}
```

```
void code_1502(int *A, int *B)
{
    // Code 1502 is emitted when runtime pointer
    // disambiguation checks are required, and
    // an array reference has an offset that varies
    // in the loop.

    int x = 0;
    for (int i=0; i<100; i++)
    {
        A[i] = B[i + x] + 1;
        x++;                    // 'x' varies in the loop
    }
}
```

```
void code_1503(int *A, int *B, int x, int y)
{
    // Code 1503 is emitted when runtime pointer
    // disambiguation checks are required, and
    // an array reference has multiple offsets.

    for (int i=0; i<100; i++)
    {
        A[i] = B[i+x] + B[i+y] + 1;   // multiple offsets when addressing 'B': {x, y}
        A[i] = B[i+x] + B[i] + 1;     // multiple offsets when addressing 'B': {x, 0}
        A[i] = B[i+x] + B[i+x] + 1;   // this is vectorized
    }
}
```

```
void code_1504(int *A1, int *A2, int *A3, int *A4,
                int *A5, int *A6, int *A7, int *A8,
                int *A9, int *A10, int *A11, int *A12,
                int *A13, int *A14, int *A15, int *A16)
{
    // Code 1504 is emitted when too many runtime
    // pointer disambiguation checks are required.

    for (int i=0; i<100; i++)
    {
        A1[i]++;
        A2[i]++;
        A3[i]++;
        A4[i]++;
        A5[i]++;
        A6[i]++;
        A7[i]++;
        A8[i]++;
        A9[i]++;
        A10[i]++;
        A11[i]++;
        A12[i]++;
        A13[i]++;
        A14[i]++;
        A15[i]++;
        A16[i]++;
    }
}
```

```
void code_1505(int *A, int *B)
{
    // Code 1505 is emitted when runtime pointer
    // disambiguation checks are required, but are
    // too complex for the compiler to discern.

    for (int i=0; i<100; i++)
    {
        for (int j=0; j<100; j++)
        {
            for (int k=0; k<100; k++)
            {
                A[i+j-k] = B[i-j+k] + 1;
            }
        }
    }
}
```