Case study #0 (part II)

Laplacian Stencil Application
(Today : on 2D grid)
Kernel header (Laplacian.h)

#pragma once

#define XDIM 2048
#define YDIM 2048

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM]);

Execution:

Running test iteration 1 [Elapsed time : 25.4213ms]
Running test iteration 2 [Elapsed time : 10.8833ms]
Running test iteration 3 [Elapsed time : 0.807804ms]
Running test iteration 4 [Elapsed time : 0.325908ms]
Running test iteration 5 [Elapsed time : 0.307869ms]
Running test iteration 6 [Elapsed time : 0.29541ms]
Running test iteration 7 [Elapsed time : 0.298488ms]
Running test iteration 8 [Elapsed time : 0.298959ms]
Running test iteration 9 [Elapsed time : 0.298472ms]
Running test iteration 10 [Elapsed time : 0.299072ms]
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{
    #pragma omp parallel for
    for (int j = 1; j < YDIM-1; j++)
        for (int i = 1; i < XDIM-1; i++)
            Lu[i][j] =
                -4 * u[i][j]
                + u[i+1][j]
                + u[i-1][j]
                + u[i][j+1]
                + u[i][j-1];
}

Execution:

Running test iteration  1 [Elapsed time : 88.9032ms]
Running test iteration  2 [Elapsed time : 50.2971ms]
Running test iteration  3 [Elapsed time : 50.5499ms]
Running test iteration  4 [Elapsed time : 50.2705ms]
Running test iteration  5 [Elapsed time : 51.0571ms]
Running test iteration  6 [Elapsed time : 51.5478ms]
Running test iteration  7 [Elapsed time : 51.4321ms]
Running test iteration  8 [Elapsed time : 50.3991ms]
Running test iteration  9 [Elapsed time : 50.4688ms]
Running test iteration 10 [Elapsed time : 52.8201ms]
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{
    #pragma omp parallel for
    for (int j = 1; j < YDIM-1; j++)
        for (int i = 1; i < XDIM-1; i++)
            Lu[i][j] =
                -4 * u[i][j]
                + u[i+1][j]
                + u[i-1][j]
                + u[i][j+1]
                + u[i][j-1];
}
#include "Laplacian.h"

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM])
{
    #pragma omp parallel for
    for (int j = 1; j < YDIM-1; j++)
        for (int i = 1; i < XDIM-1; i++)
            Lu[i][j] =
                -4 * u[i][j] + u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1];
}

 Execution:
Running test iteration  1 [Elapsed time : 2034.53ms]
Running test iteration  2 [Elapsed time : 1814.3ms]
Running test iteration  3 [Elapsed time : 1873.85ms]
Running test iteration  4 [Elapsed time : 1779.44ms]
Running test iteration  5 [Elapsed time : 1731.12ms]
Running test iteration  6 [Elapsed time : 1809.28ms]
Running test iteration  7 [Elapsed time : 1825.35ms]
Running test iteration  8 [Elapsed time : 1725.44ms]
Running test iteration  9 [Elapsed time : 1806.62ms]
Running test iteration 10 [Elapsed time : 1882.4ms]
```cpp
#include "Timer.h"
#include "Laplacian.h"

#include <iomanip>

int main(int argc, char *argv[]) {
    float **u = new float *[XDIM];
    float **Lu = new float *[XDIM];
    for (int i = 0; i < XDIM; i++){
        u[i] = new float [YDIM];
        Lu[i] = new float [YDIM];
    }

    Timer timer;

    for(int test = 1; test <= 10; test++){
        std::cout << "Running test iteration " << std::setw(2) << test << " ";
        timer.Start();
        ComputeLaplacian(u, Lu);
        timer.Stop("Elapsed time : ");
    }

    return 0;
}
```
Kernel header (Laplacian.h)

#define XDIM 2048
#define YDIM 2048

void ComputeLaplacian(const float **u, float **Lu);

Arguments passed as double pointers
(Laplacian.cpp is largely unchanged)

Execution:
Running test iteration 1 [Elapsed time : 20.1705ms]
Running test iteration 2 [Elapsed time : 1.51735ms]
Running test iteration 3 [Elapsed time : 1.51338ms]
Running test iteration 4 [Elapsed time : 0.668702ms]
Running test iteration 5 [Elapsed time : 0.621804ms]
Running test iteration 6 [Elapsed time : 0.62804ms]
Running test iteration 7 [Elapsed time : 0.623426ms]
Running test iteration 8 [Elapsed time : 0.623373ms]
Running test iteration 9 [Elapsed time : 0.624101ms]
Running test iteration 10 [Elapsed time : 0.61673ms]
Kernel header (Laplacian.h)

```c
#pragma once

#define XDIM 16384
#define YDIM 256

void ComputeLaplacian(const float (&u)[XDIM][YDIM], float (&Lu)[XDIM][YDIM]);
```

Rectangular size, 16K x 256
(same overall size as 2K x 2K)

Execution:

<table>
<thead>
<tr>
<th>Running test iteration</th>
<th>Elapsed time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19.4975ms</td>
</tr>
<tr>
<td>2</td>
<td>0.695738ms</td>
</tr>
<tr>
<td>3</td>
<td>0.692519ms</td>
</tr>
<tr>
<td>4</td>
<td>0.692588ms</td>
</tr>
<tr>
<td>5</td>
<td>0.693134ms</td>
</tr>
<tr>
<td>6</td>
<td>0.752835ms</td>
</tr>
<tr>
<td>7</td>
<td>0.348585ms</td>
</tr>
<tr>
<td>8</td>
<td>0.299074ms</td>
</tr>
<tr>
<td>9</td>
<td>0.32255ms</td>
</tr>
<tr>
<td>10</td>
<td>0.299462ms</td>
</tr>
</tbody>
</table>
#include "Timer.h"
#include "Laplacian.h"
#include <iomanip>
#include <random>

int main(int argc, char *argv[]) {
    float **u = new float *[XDIM];
    float **Lu = new float *[XDIM];

    // Randomize allocation of minor array dimension
    std::vector<int> reorderMap;
    std::vector<int> tempMap;
    for (int i = 0; i < XDIM; i++) tempMap.push_back(i);
    std::random_device r; std::default_random_engine e(r());
    while (!tempMap.empty()) {
        std::uniform_int_distribution<int> uniform_dist(0, tempMap.size()-1);
        int j = uniform_dist(e);
        reorderMap.push_back(tempMap[j]); tempMap[j] = tempMap.back();
        tempMap.pop_back();
    }

    for (int i = 0; i < XDIM; i++) {
        u[reorderMap[i]] = new float [YDIM];
        Lu[reorderMap[i]] = new float [YDIM];
    }

    Timer timer;
    for(int test = 1; test <= 10; test++)
    {
        std::cout << "Running test iteration " << std::setw(2) << test << " ";
        timer.Start();
        ComputeLaplacian(u, Lu);
        timer.Stop("Elapsed time :");
    }
    return 0;
}
#pragma once
#define XDIM 16384
#define YDIM 256

void ComputeLaplacian(const float **u, float **Lu);

Arguments passed as double pointers
(Laplacian.cpp is largely unchanged)
(with randomized allocation)

Execution:
Running test iteration 1 [Elapsed time : 10.0235ms]
Running test iteration 2 [Elapsed time : 0.750141ms]
Running test iteration 3 [Elapsed time : 0.725621ms]
Running test iteration 4 [Elapsed time : 0.830286ms]
Running test iteration 5 [Elapsed time : 0.801024ms]
Running test iteration 6 [Elapsed time : 0.78661ms]
Running test iteration 7 [Elapsed time : 0.714213ms]
Running test iteration 8 [Elapsed time : 0.71165ms]
Running test iteration 9 [Elapsed time : 0.713606ms]
Running test iteration 10 [Elapsed time : 0.771579ms]
Practical use of SIMD in code
<table>
<thead>
<tr>
<th>Instruction Stream</th>
<th>Single</th>
<th>Multi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>SISD (Single-Core Processors)</td>
<td>SIMD (GPUs, Intel SSE/AVX extensions, ...)</td>
</tr>
<tr>
<td>Multi</td>
<td>MISD (Systolic Arrays, ...)</td>
<td>MIMD (VLIW, Parallel Computers)</td>
</tr>
</tbody>
</table>
**Single-Instruction Single-Data (Single-Core Processors)**

**Single-Instruction Multi-Data (GPUs, Intel SIMD)**

**Multi-Instruction Single-Data (Systolic Arrays,...)**

**Multi-Instruction Multi-Data (Parallel Computers)
Intel SIMD Registers (AVX-512)

- **XMM0 - XMM15**
  - 128-bit registers
  - SSE

- **YMM0 - YMM15**
  - 256-bit registers
  - AVX, AVX2

- **ZMM0 - ZMM31**
  - 512-bit registers
  - AVX-512
SSE/AVX Data Types

Operation on 32 8-bit values in one instruction!
Sandy Bridge Microarchitecture

e.g., “Port 5 pressure” when code uses too much shuffle operations
As a basis for the usage model discussed in this section, consider a simple loop shown in Example 4-7. Note that the loop runs for only four iterations. This allows a simple replacement of the code with Streaming SIMD Extensions.

For the optimal use of the Streaming SIMD Extensions that need data alignment on the 16-byte boundary, all examples in this chapter assume that the arrays passed to the routine, \( A \), \( B \), \( C \), are aligned to 16-byte boundaries by a calling routine. For the methods to ensure this alignment, please refer to the application notes for the Pentium 4 processor.

The sections that follow provide details on the coding methodologies: inlined assembly, intrinsics, C++ vector classes, and automatic vectorization.

### 4.3.1.1 Assembly

Key loops can be coded directly in assembly language using an assembler or by using inlined assembly (C-asm) in C/C++ code. The Intel compiler or assembler recognize the new instructions and registers, then directly generate the corresponding code. This model offers the opportunity for attaining greatest performance, but this performance is not portable across the different processor architectures.

#### Example 4-13. Simple Four-Iteration Loop

```c
void add(float *a, float *b, float *c)
{
    int i;
    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```
Intrinsics provide the access to the ISA functionality using C/C++ style coding instead of assembly language. Intel has defined three sets of intrinsic functions that are implemented in the Intel C++ Compiler to support the MMX technology, Streaming SIMD Extensions and Streaming SIMD Extensions 2. Four new C data types, representing 64-bit and 128-bit objects are used as the operands of these intrinsic functions. __M64 is used for MMX integer SIMD, __M128 is used for single-precision floating-point SIMD, __M128I is used for Streaming SIMD Extensions 2 integer SIMD, and __M128D is used for double precision floating-point SIMD. These types enable the programmer to choose the implementation of an algorithm directly, while allowing the compiler to perform register allocation and instruction scheduling where possible. The intrinsics are portable among all Intel architecture-based processors supported by a compiler.

The use of intrinsics allows you to obtain performance close to the levels achievable with assembly. The cost of writing and maintaining programs with intrinsics is considerably less. For a detailed description of the intrinsics and their use, refer to the Intel C++ Compiler documentation.

Example 4-14. Streaming SIMD Extensions Using Inlined Assembly Encoding

```c
void add(float *a, float *b, float *c)
{
    __asm {
        mov     eax, a
        mov     edx, b
        mov     ecx, c
        movaps  xmm0, XMMWORD PTR [eax]
        addps   xmm0, XMMWORD PTR [edx]
        movaps  XMMWORD PTR [ecx], xmm0
    }
}
```
Example 4-14. Streaming SIMD Extensions Using Inlined Assembly Encoding

```c
void add(float *a, float *b, float *c)
{
    __asm {
        mov     eax, a
        mov     edx, b
        mov     ecx, c
        movaps  xmm0, XMMWORD PTR [eax]
        addps   xmm0, XMMWORD PTR [edx]
        movaps  XMMWORD PTR [ecx], xmm0
    }
}
```

✓ Anything that *can* be done, can be coded up as inline assembly
✓ Maximum *potential* for performance accelerations
✓ Direct control over the code being generated
Example 4-14. Streaming SIMD Extensions Using Inlined Assembly Encoding

```c
void add(float *a, float *b, float *c)
{
    __asm {
        mov     eax, a
        mov     edx, b
        mov     ecx, c
        movaps  xmm0, XMMWORD PTR [eax]
        addps   xmm0, XMMWORD PTR [edx]
        movaps  XMMWORD PTR [ecx], xmm0
    }
}
```

✓ Anything that can be done, can be coded up as inline assembly
✓ Maximum potential for performance accelerations
✓ Direct control over the code being generated
✗ Impractical for all but the smallest of kernels
✗ Not portable
✗ User needs to perform register allocation (and save old registers)
✗ User needs to (expertly) schedule instructions to hide latency
Intrinsics

- A framework for generating assembly-level code without many of the drawbacks of inline assembly
  - Compiler (not programmer) takes care of register allocation
  - Compiler is able to schedule instructions to hide latencies
- Data types
  - Scalar: float, double, unsigned int ...
  - Vector: __mm128, __m128d, __m256, __m256i ...
- Intrinsic functions
  - Instruction wrappers: _mm_add_pd, _mm256_mult_pd, _mm_xor_ps, _mm_sub_ss ...
  - Macros: _mm_set1_ps, _mm256_setzero_ps ...
  - Math Wrappers: _mm_log_ps, _mm256_pow_pd ...
Example 4-9 shows the loop from Example 4-7 using intrinsics. The intrinsics map one-to-one with actual Streaming SIMD Extensions assembly code. The XMMINTRIN.H header file in which the prototypes for the intrinsics are defined is part of the Intel C++ Compiler included with the VTune Performance Enhancement Environment CD. Intrinsics are also defined for the MMX technology ISA. These are based on the __m64 data type to represent the contents of an mm register. You can specify values in bytes, short integers, 32-bit values, or as a 64-bit object.

The intrinsic data types, however, are not a basic ANSI C data type, and therefore you must observe the following usage restrictions:

- Use intrinsic data types only on the left-hand side of an assignment as a return value or as a parameter. You cannot use it with other arithmetic expressions (for example, V+W, V>>W).
- Use intrinsic data type objects in aggregates, such as unions to access the byte elements and structures; the address of an __M64 object may be also used.
- Use intrinsic data type data only with the MMX technology intrinsics described in this guide.

For complete details of the hardware instructions, see the Intel Architecture MMX Technology Programmer’s Reference Manual. For a description of data types, see the Intel® 64 and IA-32 Architectures Software Developer’s Manual.

4.3.1.3 Classes

A set of C++ classes has been defined and available in Intel C++ Compiler to provide both a higher-level abstraction and more flexibility for programming with MMX technology, Streaming SIMD Extensions and Streaming SIMD Extensions 2. These classes provide an easy-to-use and flexible interface to the intrinsic functions, allowing developers to write more natural C++ code without worrying about which intrinsic or assembly language instruction to use for a given operation. Since the intrinsic functions underlie the implementation of these C++ classes, the performance benefits can be significant.

Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```c
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```
Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```c
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

Example 4-14. Streaming SIMD Extensions Using Inlined Assembly Encoding

```c
void add(float *a, float *b, float *c)
{
    __asm {
        mov     eax, a
        mov     edx, b
        mov     ecx, c
        movaps  xmm0, XMMWORD PTR [eax]
        addps   xmm0, XMMWORD PTR [edx]
        movaps  XMMWORD PTR [ecx], xmm0
    }
}
```
Example 4-9 shows the loop from Example 4-7 using intrinsics. The intrinsics map one-to-one with actual Streaming SIMD Extensions assembly code. The XMMINTRIN.H header file in which the prototypes for the intrinsics are defined is part of the Intel C++ Compiler included with the VTune Performance Enhancement Environment CD.

Intrinsics are also defined for the MMX technology ISA. These are based on the __m64 data type to represent the contents of an xmm register. You can specify values in bytes, short integers, 32-bit values, or as a 64-bit object.

The intrinsic data types, however, are not a basic ANSI C data type, and therefore you must observe the following usage restrictions:

- Use intrinsic data types only on the left-hand side of an assignment as a return value or as a parameter. You cannot use it with other arithmetic expressions (for example, V+W, V>>W).
- Use intrinsic data type objects in aggregates, such as unions to access the byte elements and structures; the address of an __M64 object may be also used.
- Use intrinsic data type data only with the MMX technology intrinsics described in this guide.

For complete details of the hardware instructions, see the Intel Architecture MMX Technology Programmer's Reference Manual. For a description of data types, see the Intel® 64 and IA-32 Architectures Software Developer's Manual.

4.3.1.3 Classes

A set of C++ classes has been defined and available in Intel C++ Compiler to provide both a higher-level abstraction and more flexibility for programming with MMX technology, Streaming SIMD Extensions and Streaming SIMD Extensions 2. These classes provide an easy-to-use and flexible interface to the intrinsic functions, allowing developers to write more natural C++ code without worrying about which intrinsic or assembly language instruction to use for a given operation. Since the intrinsic functions underlie the implementation of these C++ classes, the performance benefits of using intrinsics can be realized.

Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```c
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```
Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```c
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

✓ Almost as flexible as inline assembly
✓ Somewhat portable
✓ Compiler takes care of register allocation (and spill, if needed)
✓ Compiler will shuffle & schedule instructions to best hide latencies
✓ Relatively easy migration
Example 4-9 shows the loop from Example 4-7 using intrinsics. The intrinsics map one-to-one with actual Streaming SIMD Extensions assembly code. The XMMINTRIN.H header file in which the prototypes for the intrinsics are defined is part of the Intel C++ Compiler included with the VTune Performance Enhancement Environment CD.

Intrinsics are also defined for the MMX technology ISA. These are based on the __m64 data type to represent the contents of an mm register. You can specify values in bytes, short integers, 32-bit values, or as a 64-bit object. The intrinsic data types, however, are not a basic ANSI C data type, and therefore you must observe the following usage restrictions:

- Use intrinsic data types only on the left-hand side of an assignment as a return value or as a parameter. You cannot use it with other arithmetic expressions (for example, V+W, V>>W).
- Use intrinsic data type objects in aggregates, such as unions to access the byte elements and structures; the address of an __M64 object may be also used.
- Use intrinsic data type data only with the MMX technology intrinsics described in this guide.

For complete details of the hardware instructions, see the Intel Architecture MMX Technology Programmer's Reference Manual. For a description of data types, see the Intel® 64 and IA-32 Architectures Software Developer's Manual.

4.3.1.3 Classes

A set of C++ classes has been defined and available in Intel C++ Compiler to provide both a higher-level abstract ion and more flexibility for programming with MMX technology, Streaming SIMD Extensions and Streaming SIMD Extensions 2. These classes provide an easy-to-use and flexible interface to the intrinsic functions, allowing developers to write more natural C++ code without worrying about which intrinsic or assembly language instruction to use for a given operation. Since the intrinsic functions underlie the implementation of these C++ classes, the performance benefits are preserved.

Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```c
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
  __m128 t0, t1;
  t0 = _mm_load_ps(a);
  t1 = _mm_load_ps(b);
  t0 = _mm_add_ps(t0, t1);
  _mm_store_ps(c, t0);
}
```

✓ Almost as flexible as inline assembly
✓ Somewhat portable
✓ Compiler takes care of register allocation (and spill, if needed)
✓ Compiler will shuffle & schedule instructions to best hide latencies
✓ Relatively easy migration
✗ Coding large kernels is still challenging and bug-prone
✗ Un-natural notation (vs. C++ expressions and operators)
✗ SSE code is similar to AVX code, but different enough so that 2 distinct versions must be written
✗ Vector code looks very different than scalar code
CODING FOR SIMD ARCHITECTURES

Example 4-11 shows the code for automatic vectorization for the simple four-iteration loop (from Example 4-7).

Compile this code using the -QAX and -QRESTRICT switches of the Intel C++ Compiler, version 4.0 or later.

The RESTRICT qualifier in the argument list is necessary to let the compiler know that there are no other aliases to the memory to which the pointers point. In other words, the pointer for which it is used, provides the only means of accessing the memory in question in the scope in which the pointers live. Without the restrict qualifier, the compiler will still vectorize this loop using runtime data dependence testing, where the generated code dynamically selects between sequential or vector execution of the loop, based on overlap of the parameters (See documentation for the Intel C++ Compiler). The restrict keyword avoids the associated overhead altogether. See Intel C++ Compiler documentation for details.

4.4 STACK AND DATA ALIGNMENT

To get the most performance out of code written for SIMD technologies data should be formatted in memory according to the guidelines described in this section. Assembly code with an unaligned accesses is a lot slower than an aligned access.

4.4.1 Alignment and Contiguity of Data Access Patterns

The 64-bit packed data types defined by MMX technology, and the 128-bit packed data types for Streaming SIMD Extensions and Streaming SIMD Extensions 2 create more potential for misaligned data accesses. The data access patterns of many algorithms are inherently misaligned when using MMX technology and Streaming SIMD Extensions. Several techniques for improving data access, such as padding, organizing data elements into arrays, etc. are described below. SSE3 provides a special...

Example 4-17. Automatic Vectorization for a Simple Loop

```c
void add (float *restrict a,
          float *restrict b,
          float *restrict c)
{
    int i;
    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```
Example 4-17. Automatic Vectorization for a Simple Loop

```c
void add (float *restrict a,
    float *restrict b,
    float *restrict c)
{
    int i;
    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

✓ Minimal effort required (assuming it works ...)
✓ Development of SIMD code is no different than scalar code
✓ Ability to use complex C++ expressions
✓ Larger kernels are easier to tackle
Example 4-11 shows the code for automatic vectorization for the simple four-iteration loop (from Example 4-7).

Compile this code using the -QAX and -QRESTRICT switches of the Intel C++ Compiler, version 4.0 or later. The RESTRICT qualifier in the argument list is necessary to let the compiler know that there are no other aliases to the memory to which the pointers point. In other words, the pointer for which it is used, provides the only means of accessing the memory in question in the scope in which the pointers live. Without the restrict qualifier, the compiler will still vectorize this loop using runtime data dependence testing, where the generated code dynamically selects between sequential or vector execution of the loop, based on overlap of the parameters (See documentation for the Intel C++ Compiler). The restrict keyword avoids the associated overhead altogether. See Intel C++ Compiler documentation for details.

4.4 STACK AND DATA ALIGNMENT

To get the most performance out of code written for SIMD technologies data should be formatted in memory according to the guidelines described in this section. Assembly code with an unaligned accesses is a lot slower than an aligned access.

4.4.1 Alignment and Contiguity of Data Access Patterns

The 64-bit packed data types defined by MMX technology, and the 128-bit packed data types for Streaming SIMD Extensions and Streaming SIMD Extensions 2 create more potential for misaligned data accesses. The data access patterns of many algorithms are inherently misaligned when using MMX technology and Streaming SIMD Extensions. Several techniques for improving data access, such as padding, organizing data elements into arrays, etc. are described below. SSE3 provides a special

Example 4-17. Automatic Vectorization for a Simple Loop

```c
void add (float *restrict a,
          float *restrict b,
          float *restrict c)
{
    int i;
    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

✓ Minimal effort required (assuming it works ...)  ❌ In practice it can be very challenging to achieve efficiency comparable to assembly/intrinsics
✓ Development of SIMD code is no different than scalar code   ❌ Compilers are very conservative when vectorizing, for the risk of jeopardizing scalar equivalence
✓ Ability to use complex C++ expressions   ❌ The no-aliasing restriction might run contrary to the spirit of certain kernels
✓ Larger kernels are easier to tackle
The performance of applications using this methodology can approach that of one using the intrinsics. Further details on the use of these classes can be found in the Intel C++ Class Libraries for SIMD Operations User's Guide, order number 693500.

Example 4-10 shows the C++ code using a vector class library. The example assumes the arrays passed to the routine are already aligned to 16-byte boundaries.

Here, fvec.h is the class definition file and F32vec4 is the class representing an array of four floats. The + and = operators are overloaded so that the actual Streaming SIMD Extensions implementation in the previous example is abstracted out, or hidden, from the developer. Note how much more this resembles the original code, allowing for simpler and faster programming.

Again, the example is assuming the arrays, passed to the routine, are already aligned to 16-byte boundary.

4.3.1.4 Automatic Vectorization

The Intel C++ Compiler provides an optimization mechanism by which loops, such as in Example 4-7 can be automatically vectorized, or converted into Streaming SIMD Extensions code. The compiler uses similar techniques to those used by a programmer to identify whether a loop is suitable for conversion to SIMD. This involves determining whether the following might prevent vectorization:

1. The layout of the loop and the data structures used
2. Dependencies amongst the data accesses in each iteration and across iterations

Once the compiler has made such a determination, it can generate vectorized code for the loop, allowing the application to use the SIMD instructions.

The caveat to this is that only certain types of loops can be automatically vectorized, and in most cases user interaction with the compiler is needed to fully enable this.

Example 4-16. C++ Code Using the Vector Classes

```cpp
#include <fvec.h>
void add(float *a, float *b, float *c)
{
    F32vec4 *av=(F32vec4 *) a;
    F32vec4 *bv=(F32vec4 *) b;
    F32vec4 *cv=(F32vec4 *) c;
    *cv=*av + *bv;
}
```
The performance of applications using this methodology can approach that of one using the intrinsic. Further details on the use of these classes can be found in the Intel C++ Class Libraries for SIMD Operations User's Guide, order number 693500.

Example 4-10 shows the C++ code using a vector class library. The example assumes the arrays passed to the routine are already aligned to 16-byte boundaries. Here, fvec.h is the class definition file and F32vec4 is the class representing an array of four floats. The I+J and I=J operators are overloaded so that the actual Streaming SIMD Extensions implementation in the previous example is abstracted out, or hidden, from the developer. Note how much more this resembles the original code, allowing for simpler and faster programming.

Again, the example is assuming the arrays, passed to the routine, are already aligned to 16-byte boundaries.

4.3.1.4 Automatic Vectorization

The Intel C++ Compiler provides an optimization mechanism by which loops, such as in Example 4-7 can be automatically vectorized, or converted into Streaming SIMD Extensions code. The compiler uses similar techniques to those used by a programmer to identify whether a loop is suitable for conversion to SIMD. This involves determining whether the following might prevent vectorization:

- The layout of the loop and the data structures used
- Dependencies amongst the data accesses in each iteration and across iterations

Once the compiler has made such a determination, it can generate vectorized code for the loop, allowing the application to use the SIMD instructions.

The caveat to this is that only certain types of loops can be automatically vectorized, and in most cases user interaction with the compiler is needed to fully enable this.

Example 4-16. C++ Code Using the Vector Classes

```cpp
#include <fvec.h>
void add(float *a, float *b, float *c)
{
    F32vec4 *av=(F32vec4 *) a;
    F32vec4 *bv=(F32vec4 *) b;
    F32vec4 *cv=(F32vec4 *) c;
    *cv=*av + *bv;
}
```

✓ Fewer visual differences between vector and scalar code
✓ Ability to use complex C++ expressions (assuming wrapper types have been overloaded)
✓ Easy transition to different vector widths
Example 4-16. C++ Code Using the Vector Classes

```c
#include <fvec.h>
void add(float *a, float *b, float *c)
{
    F32vec4 *av=(F32vec4 *) a;
    F32vec4 *bv=(F32vec4 *) b;
    F32vec4 *cv=(F32vec4 *) c;
    *cv=*av + *bv;
}
```

✓ Fewer visual differences between vector and scalar code
✓ Ability to use complex C++ expressions (assuming wrapper types have been overloaded)
✓ Easy transition to different vector widths
✗ Heavy dependence on the compiler for eliminating temporaries (but it typically does a really good job at it)
✗ Limited to the semantics of the built-in vector wrapper classes (but we are free to extend those)
✗ Risk of more bloated executable code than by using intrinsics