Processor-Specific Optimization

Theoretically, .NET developers should never be concerned with optimizations tailored to a specific processor or instruction set. After all, the purpose of IL and JIT compilation is to allow managed applications to run on any hardware that has the .NET Framework installed, and to remain indifferent to operating system bitness, processor features, and instruction sets. However, squeezing the last bits of performance from managed applications may require reasoning at the assembly language level, as we have seen throughout this book. At other times, understanding processor-specific features is a first step for even more significant performance gains.

In this short section we will review some examples of optimization tailored to specific processor features, optimization that might work well on one machine and break on another. We focus mostly on Intel processors, especially the Nehalem, Sandy Bridge and Ivy Bridge families, but most of the guidance will also be relevant to AMD processors. Because these optimizations are perilous and sometimes unrepeatable, you should not use these examples as definitive guidance, but only as motivation for reaping more performance from your applications.

Single Instruction Multiple Data (SIMD)

Data-level parallelism, also known as *Single Instruction Multiple Data* (SIMD), is a feature of modern processors that enables the execution of a single instruction on a large set of data (larger than the machine word). The de-facto standard for SIMD instruction sets is SSE (Streaming SIMD Extensions), used by Intel processors since Pentium III. This instruction set adds new 128-bit registers (with the `XMM` prefix) as well as instructions that can operate on them. Recent Intel processors introduced *Advanced Vector Extensions* (AVX), which is an extension of SSE that offers 256-bit registers and even more SIMD instructions. Some examples of SSE instructions include:

- Integer and floating-point arithmetic
- Comparisons, shuffling, data type conversion (integer to floating-point)
- Bitwise operations
- Minimum, maximum, conditional copies, CRC32, population count (introduced in SSE4 and later)

You might be wondering whether instructions operating on these "new" registers are slower than their standard counterparts. If that were the case, any performance gains would be deceiving. Fortunately, that is not the case. On Intel i7 processors, a floating-point addition (`FADD`) instruction on 32-bit registers has throughput of one instruction per cycle and latency of 3 cycles. The equivalent `ADDPS` instruction on 128-bit registers also has throughput of one instruction per cycle and latency of 3 cycles.

LATENCY AND THROUGHPUT

Latency and throughput are common terms in general performance measurements, but especially so when discussing the "speed" of processor instructions:

- An instruction's latency is the time (usually measured in clock cycles) it takes to execute one instance of the instruction from start to finish.
- An instruction's throughput is the number of instructions of the same type that can be executed in a unit of time (usually measured in clock cycles).

If we say that `FADD` has latency of 3 cycles, it means that a single FADD operation will take 3 cycles to complete. If we say that FADD has throughput of one instruction per cycle, it means that by issuing multiple instances of `FADD` concurrently the processor is able to sustain an execution rate of one instruction per cycle, which will like require three of these instructions to execute concurrently.

Very often, an instruction's throughput is significantly better than its latency because processors can issue and execute multiple instructions in parallel (we will return to this subject later).

Using these instructions in high-performance loops can provide up to 8× performance gains compared to naïve sequential programs that operate on a single floating-point or integer value at a time. For example, consider the following (admittedly trivial) code:

```
//Assume that A, B, C are equal-size float arrays
for (int i = 0; i < A.length; ++i) {
  C[i] = A[i] + B[i];
}
```

The standard code emitted by the JIT in this scenario is the following:

```
; ESI has A, EDI has B, ECX has C, EDX is the iteration variable
xor edx,edx
cmp dword ptr [esi + 4],0
jle END_LOOP
NEXT_ITERATION:
fld dword ptr [esi + edx*4 + 8]            ; load A[i], no range check
cmp edx,dword ptr [edi + 4]             ; range check before accessing B[i]
jae OUT_OF_RANGE
fadd dword ptr [edi + edx*4 + 8]            ; add B[i]
cmp edx,dword ptr [ecx + 4]             ; range check before accessing C[i]
jae OUT_OF_RANGE
fstp dword ptr [ecx + edx*4 + 8]            ; store into C[i]
inc edx
cmp dword ptr [esi + 4],edx             ; are we done yet?
jg NEXT_ITERATION
END_LOOP:
```

Each loop iteration performs a single `FADD` instruction that adds two 32-bit floating-point numbers. However, by using 128-bit SSE instructions, four iterations of the loop can be issued at a time, as follows (the code below performs no range checks and assumes that the number of iterations is equally divisible by 4):

```
xor edx, edx
NEXT_ITERATION:
movups xmm1, xmmword ptr [edi + edx*4 + 8]             ; copy 16 bytes from B to xmm1
movups xmm0, xmmword ptr [esi + edx*4 + 8]             ; copy 16 bytes from A to xmm0
addps xmm1, xmm0             ; add xmm0 to xmm1 and store the result in xmm1
movups xmmword ptr [ecx + edx*4 + 8], xmm1             ; copy 16 bytes from xmm1 to C
add edx, 4             ; increase loop index by 4
```

```
cmp edx, dword ptr [esi + 4]
jg NEXT_ITERATION
```

On an AVX processor, we could move even more data around in each iteration (with the 256-bit `YMM*` registers), for an even bigger performance improvement:

```
xor edx, edx
NEXT_ITERATION:
vmovups ymm1, ymmword ptr [edi + edx*4 + 8]             ; copy 32 bytes from B to ymm1
vmovups ymm0, ymmword ptr [esi + edx*4 + 8]             ; copy 32 bytes from A to ymm0
vaddps ymm1, ymm1, ymm0                                 ; add ymm0 to ymm1 and store the result in ymm
vmovups ymmword ptr [ecx + edx*4 + 8], ymm1             ; copy 32 bytes from ymm1 to C
add edx, 8              ; increase loop index by 8
cmp edx, dword ptr [esi + 4]
jg NEXT_ITERATION
```

Note The SIMD instructions used in these examples are only the tip of the iceberg. Modern applications and games use SIMD instructions to perform complex operations, including scalar product, shuffling data around in registers and memory, checksum calculation, and many others. Intel's AVX portal is a good way to learn thoroughly what AVX can offer: http://software.intel.com/en-us/avx/

The JIT compiler uses only a small number of SSE instructions, even though they are available on practically every processor manufactured in the last 10 years. Specifically, the JIT compiler uses the SSE `MOVQ` instruction to copy medium-sized structures through the `XMM*` registers (for large structures, `REP MOVS` is used instead), and uses SSE2 instructions for floating point to integer conversion and other corner cases. The JIT compiler *does not* auto-vectorize loops by unifying iterations, as we did manually in the preceding code listings, whereas modern C++ compilers (including Visual Studio 2012) do.

Unfortunately, C# doesn't offer any keywords for embedding inline assembly code into your managed programs. Although you could factor out performance-sensitive parts to a C++ module and use .NET interoperability to access it, this is often clumsy. There are two other approaches for embedding SIMD code without resorting to a separate module.

A brute-force way to run arbitrary machine code from a managed application (albeit with a light interoperability layer) is to dynamically emit the machine code and then call it. The `Marshal.GetDelegateForFunctionPointer` method is key, as it returns a managed delegate pointing to an unmanaged memory location, which may contain arbitrary code. The following code allocates virtual memory with the EXECUTE_READWRITE page protection, which enables us to copy code bytes into memory and then execute them. The result, on an Intel i7-860 CPU, is a more than 2× improvement in execution time!

```
[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate void VectorAddDelegate(float[] C, float[] B, float[] A, int length);

[DllImport("kernel32.dll", SetLastError = true)]
static extern IntPtr VirtualAlloc(
  IntPtr lpAddress, UIntPtr dwSize, IntPtr flAllocationType, IntPtr flProtect);

//This array of bytes has been produced from the SSE assembly version - it is a complete
//function that accepts four parameters (three vectors and length) and adds the vectors
byte[] sseAssemblyBytes = { 0x8b, 0x5c, 0x24, 0x10, 0x8b, 0x74, 0x24, 0x0c, 0x8b, 0x7c, 0x24,
    0x08, 0x8b, 0x4c, 0x24, 0x04, 0x31, 0xd2, 0x0f, 0x10, 0x0c, 0x97,
    0x0f, 0x10, 0x04, 0x96, 0x0f, 0x58, 0xc8, 0x0f, 0x11, 0x0c, 0x91,
    0x83, 0xc2, 0x04, 0x39, 0xda, 0x7f, 0xea, 0xc2, 0x10, 0x00 };
IntPtr codeBuffer = VirtualAlloc(
    IntPtr.Zero,
    new UIntPtr((uint)sseAssemblyBytes.Length),
    0x1000 | 0x2000, //MEM_COMMIT | MEM_RESERVE
    0x40 //EXECUTE_READWRITE
);
Marshal.Copy(sseAssemblyBytes, 0, codeBuffer, sseAssemblyBytes.Length);
VectorAddDelegate addVectors = (VectorAddDelegate)
  Marshal.GetDelegateForFunctionPointer(codeBuffer, typeof(VectorAddDelegate));
//We can now use 'addVectors' to add vectors!
```

A completely different approach, which unfortunately isn't available on the Microsoft CLR, is extending the JIT compiler to emit SIMD instructions. This is the approach taken by *Mono.Simd*. Managed code developers who use the Mono .NET runtime can reference the Mono.Simd assembly and use JIT compiler support that converts operations on types such as `Vector16b` or `Vector4f` to the appropriate SSE instructions. For more information about Mono.Simd, see the official documentation at http://docs.go-mono.com/index.aspx?link=N:Mono.Simd.

Instruction-Level Parallelism

Unlike data-level parallelism, which relies on specific instructions to operate on larger chunks of data at a time, *instruction-level parallelism* (ILP) is a mechanism that executes several instructions simultaneously on the same processor. Modern processors have a deep pipeline with several types of execution units, such as a unit for accessing memory, a unit for performing arithmetic operations, and a unit for decoding CPU instructions. Pipelines enable the execution of multiple instructions to overlap as long as they are not competing for the same parts of the pipeline and as long as there are no *data dependencies* between them. Data dependencies arise when an instruction requires the result of another instruction that executes before it; for example, when an instruction reads from a memory location to which the previous instruction has written.

Note Instruction-level parallelism is *not* related to parallel programming, which we discussed in Chapter 6. When using parallel programming APIs the application runs multiple threads on multiple processors. Instruction-level parallelism enables a single thread, on a single processor, to execute multiple instructions at once. Unlike parallel programming, ILP is more difficult to control and depends heavily on program optimization.

In addition to pipelining, processors employ so-called *superscalar execution*, which uses multiple redundant units on the same processor to perform multiple operations of the same type at once. Furthermore, to minimize the impact of data dependencies on parallel instruction execution, processors will execute instructions out of their original order as long as it does not violate any data dependencies. By adding *speculative execution* (primarily by attempting to guess which side of a branch is going to be taken, but by other means as well), the processor is very likely to be able to execute additional instructions even if the next instruction in the original program order cannot be executed because of a data dependency.

Optimizing compilers are renowned for organizing the instruction sequence to maximize instruction level parallelism. The JIT compiler does not do a particularly good job, but the out-of-order execution capabilities of modern processors may offset it. However, poorly-specified programs can affect performance considerably by introducing unnecessary data dependencies—especially into loops—thus limiting instruction-level parallelism.

Consider the following three loops:

```
for (int k = 1; k < 100; ++k) {
    first[k] = a * second[k] + third[k];
}
```

```
for (int k = 1; k < 100; ++k) {
    first[k] = a * second[k] + first[k - 1];
}
for (int k = 1; k < 100; ++k) {
    first[k] = a * first[k - 1] + third[k];
}
```

We executed these loops on one of our test machines with arrays of 100 integers, for a million iterations every time. The first loop ran for ~ 190ms, the second for ~ 210ms, and the third for ~ 270ms. This is a substantial performance difference that stems from instruction-level parallelism. The first loop's iterations don't have any data dependencies—multiple iterations can be issued on the processor in any order and execute concurrently in the processor's pipeline. The second loop's iterations introduce a data dependency—to assign `first[k]`, the code depends on `first[k-1]`. However, at least the multiplication (which must complete before the addition takes place) can be issued without data dependencies. In the third loop, the situation is as bad as it gets: even the multiplication can't be issued without waiting for the data dependency from the previous iteration.

Another example is finding the maximum value in an array of integers. In a trivial implementation, each iteration depends on the currently established maximum from the previous iteration. Curiously, we can apply here the same idea encountered in Chapter 6—aggregation and then summation over local results. Specifically, finding the maximum of the entire array is equivalent to finding the maxima over the even and odd elements, and then performing one additional operation to find the global maximum. Both approaches are shown below:

```
//Naïve algorithm that carries a dependency from each loop iteration to the next
int max = arr[0];
for (int k = 1; k < 100; ++k) {
  max = Math.Max(max, arr[k]);
}

//ILP-optimized algorithm, which breaks down some of the dependencies such that within the
//loop iteration, the two lines can proceed concurrently inside the processor
int max0 = arr[0];
int max1 = arr[1];
for (int k = 3; k < 100; k += 2) {
  max0 = Math.Max(max0, arr[k-1]);
  max1 = Math.Max(max1, arr[k]);
}
int max = Math.Max(max0, max1);
```

Unfortunately, the CLR JIT compiler undermines this particular optimization by emitting less-than-optimal machine code for the second loop. In the first loop, the important values fit in registers—`max` and `k` are stored in registers. In the second loop, the JIT compiler isn't able to fit all the values in registers; if `max1` or `max0` are placed in memory instead, the loop's performance degrades considerably. The corresponding C++ implementation provides the expected performance gains—the first unrolling operation improves execution time by a factor of two, and unrolling again (using four local maxima) shaves off another 25%.

Instruction-level parallelism can be combined with data-level parallelism. Both examples considered here (the multiply-and-add loop and the maximum calculation) can benefit from using SIMD instructions for additional speedups. In the case of maxima, the `PMAXSD` SSE4 instruction operates on two sets of four packed 32-bit integers and finds the respective maxima of each pair of integers in the two sets. The following code (using the Visual C++ intrinsics from `<smmintrin.h>`) runs 3× faster than the previously best version and 7× faster than the naïve original:

```
__m128i max0 = *(__m128i*)arr;
for (int k = 4; k < 100; k += 4) {
  max0 = _mm_max_epi32(max0, *(__m128i*)(arr + k)); //Emits PMAXSD
}
int part0 = _mm_extract_epi32(max0, 0);
int part1 = _mm_extract_epi32(max0, 1);
int part2 = _mm_extract_epi32(max0, 2);
int part3 = _mm_extract_epi32(max0, 3);
int finalmax = max(part0, max(part1, max(part2, part3)));
```

When you minimize data dependencies to gain from instruction-level parallelism, the data-level parallelization (sometimes called *vectorization*) often turns up instantaneously for even bigger performance wins.

MANAGED VS. UNMANAGED CODE

A common concern voiced by .NET opponents is that the managed aspects of the CLR introduce a performance cost and render unviable the development of high-performance algorithms using C#, the .NET Framework, and the CLR. Throughout this book, and even in this chapter, we've seen several performance pitfalls that you must be aware of if you want to squeeze every bit of performance from your managed applications. Unfortunately, there will *always* be cases where unmanaged code (written in C++, C, or even hand-crafted assembly) will have better performance than its managed counterpart.

We do not intend to analyze and categorize every example off the Web where a C++ algorithm was demonstrated to be somewhat more efficient than its C# version. Still, there are some common themes that arise more often than others:

- Strictly CPU-bound numeric algorithms tend to run faster in C++, even after applying specific optimizations in C#. The reasons tend to fluctuate between array bounds checks (which the JIT compiler optimizes away only in some cases, and only for single-dimensional arrays), SIMD instructions employed by C++ compilers, and other optimizations at which C++ compilers excel such as sophisticated inlining and smart register allocation.
- Certain memory management patterns are detrimental to the GC's performance (as we have seen in Chapter 4). At times, C++ code can get memory management "right" by using pooling or reusing unmanaged memory obtained from other sources, where .NET code would struggle.
- C++ code enjoys more direct access to Win32 APIs, and does not require interoperability support, such as parameter marshaling and thread state transitions (discussed in Chapter 8). High-performance applications that exhibit a chatty interface with the operating system may run slower in .NET due to this interoperability layer.

David Piepgrass's excellent CodeProject article, "Head-to-head benchmark: C++ vs. NET" (available at http://www.codeproject.com/Articles/212856/Head-to-head-benchmark-Csharp-vs-NET), busts some of the misconceptions around managed code performance. For example, Piepgrass demonstrates that .NET collections are much faster in some cases than their C++ STL equivalents; the same applies to reading file data line-by-line using `ifstream` vs. `StreamReader`. On the other hand, some of his benchmarks emphasize the deficiencies that still exist in the 64-bit JIT compiler, and the lack of SIMD intrinsics on the CLR (which we discussed earlier) is another contributing factor to C++'s advantage.