Introduction

Case Study and Example

This case study demonstrates embedded signal processing where Intel IPP (or Intel MKL) is used in order to process the signal, and where Intel Cilk Plus is used to target multiple cores. The example consists of two programs, (1) an optional signal source shortly introduced in the section titled Artificial Signal Source, and (2) a program that reads and processes the signal. For the latter program, any signal source (not only the artificial signal source) can be used in order to be processed (and displayed).

```
unsigned long start = tick();

for (size_t i = 1, j = 0; i <= nsteps; ++i) {
    const T y = noise<T>(std::sin, level, x);
    std::cout << x << ' ' << y << std::endl; // flush via endl
    const double d = seconds(start, tick());
    const size_t m = static_cast<size_t>(d * rate + 0.5), n = i - j;
    const size_t t = m < n ? (n - m) : (m - n);
    if (t <= tol) {
        const double s = static_cast<double>(n) / rate - d;
        if (0 < s) msleep(static_cast<size_t>(1000.0 * s + 0.5));
        else if (0 > s) msleep(static_cast<size_t>(1000.0 * s - 0.5));
        else if (0 == s) start = tick();
    }
    else { // missed the req. signal rate
        start = tick();
        j = i;
    }
}
```

Figure 1: Only the standard input/output channels (console) are required in order to connect the signal, process it, and display the result. With SSH on the target, the signal source and the display can run on separate systems. The processed signal is displayed and monitored with Gnuplot* for illustration purposes.

All code snippets in this case study are presented as type-agnostic C++ code. This is not essential for capturing the idea, but it will help to also serve an audience who is used to thinking in terms of C++.

Artificial Signal Source

In order to simulate a typical use case of an embedded system, a separate program is used to generate and deliver input data that is usually acquired from a sensor attached to the embedded system.

```
unsigned long start = tick();

for (size_t i = 1, j = 0; i <= nsteps; ++i) {
    const T y = noise<T>(std::sin, level, x);
    std::cout << x << ' ' << y << std::endl; // flush via endl
    const double d = seconds(start, tick());
    const size_t m = static_cast<size_t>(d * rate + 0.5), n = i - j;
    const size_t t = m < n ? (n - m) : (m - n);
    if (t <= tol) {
        const double s = static_cast<double>(n) / rate - d;
        if (0 < s) msleep(static_cast<size_t>(1000.0 * s + 0.5));
        else if (0 > s) msleep(static_cast<size_t>(1000.0 * s - 0.5));
        else if (0 == s) start = tick();
    }
    else { // missed the req. signal rate
        start = tick();
        j = i;
    }
}
```

Figure 2: The artificial signal is a series of samples \((x, y)\) with \(y = \sin(x)\). The output is effectively unbuffered in order to keep the signal rate (Hz). To avoid drifts in the rate, the time to sleep between steps is determined on an absolute basis (tick, seconds, and msleep are implementation details in this article).
The buffer size not only manages call-overhead, but also the latency. Moreover, to process a signal it often needs an algorithmic “window size”. Also when using a library, the library functions are usually non-elemental in order to apply SIMD (or multicore).

Signal Processing

Intel® Integrated Performance Primitives (Intel® IPP)

Intel IPP is a C library with ready-to-use functions from a variety of domains covering a broad range of data types. Intel IPP focuses on in-core optimizations such as cache-blocking, Intel® Streaming SIMD Extensions (Intel® SSE), and continuously adopts new instruction set extensions. Prominent capabilities are the upcoming support for Intel® Advanced Vector Instructions 2 (Intel® AVX 2).

Figure 4: The level of noise in the plotted noisy signal is four times higher than the amplitude of the original Sinus (see source code in Figure 2). Without the Sinus plotted, it is hard to identify the originating function.

Pipeline Pattern

This case study parallelizes the stages of a signal processing application on a per-function level. Parallelism at this coarse-grained level allows the developer to start using multiple cores without the need to parallelize each signal processing algorithm. The example however uses only one signal processing stage but involves the input acquisition and the output into the pipeline. In order to speedup with multiple cores, it is likely required to involve more processing steps and more demanding computations. Have a look at the Parallelization section to go beyond this function-level parallelism.

Figure 5: The signal processing pipeline consists of three stages: (1) the signal reader parses \((x, y)\) values line-wise from standard input into its own dedicated buffer \(1\), (2) the actual processing stage of the signal operates out of place on buffer \(2\) and \(3\), whereas the final print stage \(4\) reads its own buffer \(4\) and prints to standard output.

```cpp
template<typename T> T noise(T (*f)(T), T level, T x)
{
    static const double scale = 1.0 / RAND_MAX;
    const double y = static_cast<double>(f(x));
    const double r = scale * static_cast<double>(2 * std::rand() - RAND_MAX);
    return static_cast<T>(y * level + r);
}
```

Figure 6: The pipeline indirectly assigns one of four buffers to each stage. Here, the quad buffering consists of two quad-buffers \(x\) and \(y\) for each component of the signal. At each cycle of the loop, the stage’s destination buffer is rotated by one position within the ring buffer (called “stage”) to become the source of the next stage.

```cpp
struct size_t i, j; 

for(size_t i = 0; i < nsteps && (0 < stage[1].n || 0 == i); i++)
    read_signal(size, x = stage[0].i, y = stage[0].i, stage[0].n);

process_signal(stage[1].n, x = stage[1].i, y = stage[1].i); 

print_signal(stage[3].n, x = stage[3].i, y = stage[3].i, std::cout);

std::rotate(stage, stage + 4 - 1, stage + 4); // quad-buffering
```

There are also dynamic link libraries available that provide internal multithreading based on OpenMP®.

Signal Processing (1d)

- Transforms (e.g., Wavelet)
- Convolution / Correlation
- Filtering (e.g., IIR, FIR)
- Statistics

Image Processing (2d)

- Transforms (e.g., rotation)
- Non-linear Filter (e.g., noise)
- FFT, DCT, DCT
- Statistics

Color Conversion

- Color space conversion
- Pattern e.g. Bayer
- Brightness / Contrast
- Resampling

More Domains…

- Vector / Matrix
- Integrity / Compression / Cryptography
- More Domains…

Figure 7: Signal processing with e.g., filters, transforms, and statistics is one of the main library domains of Intel IPP. In contrast to signal processing, image processing (another domain) allows to handle two-dimensional data.

Intel IPP is supplied as a sequential library \(^2\) (library name postfix is ".") for “linear”) because of the higher efficiency in case of smaller data sets, latency-constraint applications, or for better control via application-level threading. In order to support multithreading, Intel IPP is fully thread-safe.

The main sample collection of Intel IPP (extra download) contains an “advanced usage” category with an example driver along with an application requesting this driver’s service. Such an example may help to develop code that runs in kernel mode. Intel IPP provides libraries that are not position-independent (“nonpic”) in order to support code that runs in kernel mode (ring 0).

```cpp
int main(int argc, char* argv[])
{
    ippInit();
    // Start using Intel IPP!
}
```

Figure 8: Intel IPP requires calling ippInit prior to any other function of Intel IPP in order to pick the best available code path. A dynamic link library (".so", ".dll") allows for an initialization step by capturing the “load event” of the library. In contrast, static linkage takes code (according to a call) out of a static library (".a", ".lib”). However, in order to be flexible ippInit should be called regardless of using dynamic or static linkage.

The function names in Intel IPP follow a scheme that consists of a prefix indicating the library domain (e.g., “ipps” for signal processing) as well as a postfix that indicates the data types involved (e.g., “f32” in case of single-precision floating point). C language call-convention and linkage
elemental library function may not be vectorized within the user’s loop. The number of elements) and eventually serves algorithmic needs. Moreover, chunks allow making sure (e.g., ippsAutoCorrNorm_32f|64f in case of auto-correlation). Intel IPP 8.0 will introduce function arguments (library-wide consistent) in order to select a normalization method critical applications, a signal often needs to be normalized as the signal is processed (on the fly). To fit this need, Figure 9: Intel IPP is a C function library that easily binds to various programming languages. The C++ code allows exploiting the full potential of Intel MKL. Besides, MKL focuses on high workloads up to the 64-bit integer index space are supported (“ilp64”). Intel MKL focuses on high domains i.e., almost all routines support ready-to-use internal multi-threading. Massive parallelization Intel C++ Compiler

Auto-vectorization: The Intel Compiler together with Intel Cilk Plus addresses all performance dimensions of Intel Architecture processors e.g., by exploiting SIMD instruction set extensions as well as multicore. The Intel Compiler enables auto-vectorization and targets at least the SSE2 instruction set extension by default. Loops are the main targets to be vectorized. However, C and C++ provide loop constructs that allow writing code that prevents to operate on short vectors (packed data); hence the Intel Compiler can generate reports that help to adjust the code as well as taking directions (pragma lines) for effective auto-vectorization.

Adobe年轻的字体排版专家，精心制作的字体。
The directive “#pragma ivdep” ignores vector dependencies but the compiler’s analysis still aims to generate correct code (prevents vectorization in certain cases), whereas “#pragma simd” overwrites compiler heuristics up to the degree of permitting to generate incorrect code in the general case. Both directives support further attributes, and one may have a look at the entire set of directives supported by the Intel Compiler e.g., to generate streaming stores, and other optimizations.

Multicore: The Intel Compiler refers the automatic application of multiple threads as auto-parallelization. It can be seen similar to auto-vectorization but with heuristics that determine whether loop-parallelization is beneficial. However, given Intel Cilk Plus’ ease of use this may have limited application. There are two main usages of multicore that are important for embedded applications:

1. Singular, long-running or permanent tasks including background tasks. This category usually implements an event handling e.g., queuing user interactions.
2. Multiple tasks that operate on the same data. In case of data-parallelism, this includes embarrassing parallel problems but requires synchronization constructs (locks) in general.

In the first category, the task is in a 1:1-relationship to an OS thread whereas in the second category a threading runtime is about to map many tasks to a limited pool of worker threads. Note regardless of these categories, it is useful to make parallel patterns explicit e.g., the Pipeline Pattern.

```c
struct background {
    void background::work(background* self) {
        background()
        : (0), thread(work, this)
        }
    ~background() { }
    static void work(background* self); };

static void work(background* self); { i;
std::thread thread; // run a task in the background
} background task;
```

Figure 13: Singular, long-running or permanent tasks including background tasks are a good match for direct use of OS threads e.g., POSIX threads (Pthreads). In particular, C++ libraries provide an easy interface to interact with OS threads and synchronization primitives in a portable manner (see the C++11 example above). Also, Intel Threading Building Blocks (Intel® TBB) includes a rich set of primitives incl. std::thread (in case of C++11 is not available; TBB_IMPLEMENT_CPP10X). Note with Intel System Studio, Intel TBB can be built from source.

Intel® Cilk™ Plus

Multicore: Intel Cilk Plus in concert with the compiler enables forward-scaling task parallelism for C and C++ with a runtime-dynamic scheduler that maps an arbitrary number of tasks to a limited set of workers (pool of threads). This allows for composable designs where multicore parallelism can be added without supervising call chains (nested parallelism), and without oversubscribing resources. Intel Cilk Plus guarantees to unfold parallelism according to the number of workers in the pool instead of an unbound resource usage according to the number of scheduled tasks.

```c
for (size_t i = 0; i <= nsteps && (0 < stage[1].n || 0 == i); ++i) {
    cilk_spawn work(i, std::min(G, N - i));
}
```

Figure 14: Compared to Figure 6, cilk_spawn and cilk_sync have been added. In order to generate a serial version of a program that uses Intel Cilk Plus (keywords, reducers, etc.) one can compile with the “-cilk-serialize” option (with just cilk_spawn, cilk_sync, and cilk_for one can simply elide these keywords). Note that the above multibuffering approach actually allows calling read_signal, process_signal, and print_signal in any order which can be of interest with Intel Cilk Plus’ continuation-passing style.

Thinking of cilk_spawn as asynchronously launching an invocation can explain what is running concurrently before it is synced by cilk_sync. However, the worker thread that launches the first cilk_spawn also executes the spawned function (i.e., read_signal in Figure 14). This is in contrast to what a library-based threading runtime is able to achieve. The continuation however is eventually stolen by another worker (i.e., after the sequence point behind read_signal; hence the next spawn).

There are also a number of implicit synchronization points (where cilk_sync can be omitted). These are mostly obvious, but also complete the definition of the language extension in presence of exceptions (see References section for the Intel Compiler User and Reference Guide).

```c
for (size_t i = 0; i < N; i++) { /*A*/
    cilk_spawn work(i, std::min(G, N - i));
}
```

Figure 15: In situation A with only little work for each induction of i, the keyword cilk_for is introduced in code B to not only amortize the cilk_spawn, but to also employ a launch-scheme similar to a binary tree. Intel Cilk Plus allows adjusting the grain size of a cilk_for (#pragma cilk grainsize=expression) using a runtime expression. The grain size G in code C is able to accumulate more work in function D. With respect to the launch scheme the examples B and C are still not equivalent. Splitting the loop range according to a binary tree avoids accumulating the total launch overhead on a single core.

There are two notable consequences from Intel Cilk Plus’ continuation-passing style: (1) a thread continues with what is locally prepared or “hot in cache” and (2) the instructions of a scope (in the sense of C and C++) may not be executed by the same thread. A conclusion from #1 is that tuning a sequential program maps to a tuned parallel program in a more straightforward manner. In case of #2, a thread-local storage with a lifetime according to the scope cannot be used with Intel Cilk Plus. Now without a myth left, it should be also said that Intel Cilk Plus uses regular OS threads in order to perform the work. However, tasks are conceptually very lightweight user-space objects similar to fibers but this is not much different from other threading libraries such as Intel TBB.

Dynamic scheduling employs workers as needed, and the grain size varies the amount of available parallelism of a cilk_for loop (see also text of Figure 15). Of course, with cilk_spawn the number of spawned functions directly refers to the amount of parallelism that can be exploited e.g., the number of pipeline stages that can run in parallel. To summarize, setting the number of workers (see Figure 16) for different parallel sections is not the way to adjust concurrency.

---

4 T. G. Mattson, B. A. Sanders, and B. L. Massingill: Patterns for Parallel Programming

Copyright © 2013, Intel Corporation. All rights reserved.
Intel Compiler’s auto-vectorization is quite advanced and given it is effective, one would need to...vectorized (SIMD). Note Intel IPP...the outer loop is to harvest parallelism for multiple cores and where the inner loop is vectorized (SIMD). Note Intel IPP...function as shown in this code fragment uses an attribute...“kernels” are the only way to express parallelism, a not so obvious synchronization might be...domain. This usually implies being unable to scatter into neighboring positions. In cases where...function that launches over a linear range. Array notation...filter can be often expressed as a kernel function that launches over a linear range. Array notation...impede non-deterministic results in particular with floating-point data where the order of...removes these two dependencies. However, the order of processing these partitions may still vary, and hence impede non-deterministic results in particular with floating-point data where the order of...compiles, and may end up in multiple vectorized loops without code bloat. Note advanced techniques such as...are often used to combine data access and work-sharing behavior. However, a purely...domain. This usually implies being unable to scatter into neighboring positions. In cases where...function that launches over a linear range. Array notation...kernel. The kernel function as shown in this code fragment uses an attribute...filter: float* output, const float* input, const float* kernel[], size_t size, size_t n) {
    output[size-N] = 0;
    for (size_t j = 0; j < N; ++j) {
        // fir_kernel(result[size-N], input[j]:size-N, kernel[j]);
        output[size-N] += input[j]:size-N * kernel[j];
    }
}

Figure 20: As shown in Figure 19, the commented part of the inner loop is supposed to launch a function that is called an elemental function or kernel. The kernel function as shown in this code fragment uses an attribute...“kernels” are the only way to express parallelism, a not so obvious synchronization might be...filter: float* output, const float* input, const float* kernel[], size_t size, size_t n) {
    output[size-N] = 0;
    for (size_t j = 0; j < N; ++j) {
        // fir_kernel(result[size-N], input[j]:size-N, kernel[j]);
        output[size-N] += input[j]:size-N * kernel[j];
    }
}

Figure 19: Here, the inner loop of the initial implementation is interchanged with the outer loop, and the new inner loop is vectorized. Note this implementation cannot be parallelized by simply turning the for-loop into cilk_for, because multiple threads would modify the same left hand-side concurrently without synchronizing (data race).

Results

Experiment Overview

This case study compares, (1) a single-threaded implementation, (2) a C++11 based implementation with asynchronously launched std::future, and (3) a variant of (1) that uses Intel Cilk Plus keywords. All implementations use Intel IPP; hence this study is not about implementing a more optimized auto-correlation. Also, we did not focus on making unlimited parallelism available by parallelizing the signal processing algorithm itself. Instead, the pipeline pattern (often applicable for signal processing) demands an efficient threading runtime in order to extract parallelism.

---

5 See also http://www.cs.cmu.edu//afs/cs.cmu.edu/Web/People/guyb/papers/Ble90.pdf
With the current C++11 implementation (GNU® gcc 4.6) and an asynchronously launched signal into a file.

Our Intel Core i5-2640M based (dual core) system achieved a speedup of more than 1.6x for our three stage approach is able to process. To not interfere in terms of performance, we recorded given range of input rates.

Also, it is very likely that either one of the two I/O stages of our parallel approach represents the bottleneck (either parsing the signal values, or printing the result) for any given system. Moreover, our findings below were not affected by using double precision instead of single-precision.

Performance Stability

To not interfere with our experiments, the artificial signal source ran on a different system than our three-stage pipeline (signal acquisition, auto-correlation, and output). The signal source was requested to produce a signal at a rate of 10 Hz up to 15 kHz. Note our fastest signal rate was not limited by the network connection (15 kHz is approx. 700 KB/s with two numbers printed per line).

Of course, for a perfect result an implementation would reproduce a given input rate.

1. Our single-core Intel® Atom™ E660 based target system was fast enough to perfectly reproduce the given range of input rates.
2. Our Intel Core i5-2640M based system also reproduced all given input rates with both of our multicore implementations (C++11, Intel Cilk Plus).

The Intel Cilk Plus based executable with one worker was exactly on par with the single-threaded implementation. On the Intel Atom system, the parallelism due to our pipeline approach could not be exploited via instruction-level parallelism (single core); hence we did not include the E660 into the next exploration.

Throughput

Given the above results we changed our scheme to measure the maximum possible signal rate that our three stage approach is able to process. To not interfere in terms of performance, we recorded the signal into a file.

1. Our Intel Core i5-2640M based (dual core) system achieved a speedup of more than 1.6x for our coarse-grained parallelism with two of three stages being I/O stages. Note that our speedup accounts linearly for Intel® Turbo Boost Technology (2.6 vs. 3.3 GHz) in order to fairly compare multicore against a boosted single-core. However, without having measured and just based on Polack's rule (see Parallelism and Performance section) the parallelism would have saved us nearly one third of the energy [Joule] consumption.
2. With the current C++11 implementation (GNU® gcc 4.6) and an asynchronously launched std::future, one thread per each task is launched carrying overhead and resource consumption that makes it inefficient to extract small amounts of parallelism. We expect to see improvements with the availability of a thread pool. However, we believe lightweight tasks that are decoupled from actual workers are needed for forward-scaling performance.

Polack’s rule should be considered alongside the fact that performance may scale linearly with clock frequency, but energy consumption will roughly scale with the square of the clock frequency. Amdahl’s Law limits the practical use of a system that only provides performance in presence of parallelism. However, it is very attractive to prepare signal processing applications to make use of multiple cores because of possible energy savings, or to consolidate specialized hardware by loading the system with various different tasks in addition to accelerating the signal processing itself.

Software optimizations including in-core optimizations can save energy. Cache-blocking can avoid unnecessary memory loads. With multicore one can take this idea further with cache-oblivious algorithms. However, multicore parallelism by itself is able to save energy in the following way:

- 4x the die area of a microprocessor gives 2x the performance in one core, but
- 4x the performance when the same area is dedicated to 4 cores'.

Parallelism and Performance

The pipeline pattern is only able to extract a limited amount of parallelism, and the longest running stage always becomes the bottleneck. In this example, the pipeline consists of only three stages that can run in parallel where two of them are I/O. The latter can be an additional burden in terms of scalability if the I/O functionality uses locks in order to protect internal state.

It is actually more interesting in our example, that successive fork-joins (every cycle) demand an efficient threading runtime. Of course, one can try to hide the overhead with a larger buffer size. However, a parallel region that executes longer is obviously increasing the latency of the application.

Performance optimizations including in-core optimizations can save energy. Cache-blocking can avoid unnecessary memory loads. With multicore one can take this idea further with cache-oblivious algorithms. However, multicore parallelism by itself is able to save energy in the following way:

- 4x the die area of a microprocessor gives 2x the performance in one core, but
- 4x the performance when the same area is dedicated to 4 cores'.

Polack’s rule should be considered alongside the fact that performance may scale linearly with clock frequency, but energy consumption will roughly scale with the square of the clock frequency. Amdahl’s Law limits the practical use of a system that only provides performance in presence of parallelism. However, it is very attractive to prepare signal processing applications to make use of multiple cores because of possible energy savings, or to consolidate specialized hardware by loading the system with various different tasks in addition to accelerating the signal processing itself.

Summary

Intel System Studio not only provides a variety of signal processing primitives (Intel IPP and Intel MKL), but also allows developing high-performance low-latency custom code (Intel C++ Compiler with Intel Cilk Plus).

---

* Intel® VTune Amplifier™ can actually estimate the energy consumption for a given workload.

* This implies to perfectly parallelize an application.