Signal Processing with Intel® System Studio

Hans Pabst, Intel Corporation

Software for embedded systems is usually restricted to work with a specific iteration of a hardware vendor’s platform, and developers frequently give up portability due to limitations of the target hardware or the software environment. Intel System Studio allows developers to overcome these limitations by extending the reach of Intel’s developer tools. This way embedded computing can rely on common or open standards, industry standards, or (when Intel-specific) it can rely on an entire compute continuum.

Abstract
This article focuses on Intel System Studio components that deliver signal processing functionality. The development process as supported by components such as Intel® VTune™ Amplifier, Intel® Inspector, Intel® JTAG Debugger, and other process-oriented features (e.g. cross-compilation) are out of this article’s scope. We provide a case study to familiarize you with the signal processing functionality of Intel System Studio. The following components¹ are covered:

1. Intel® Integrated Performance Primitives (Intel® IPP)
2. Intel® Math Kernel Library (Intel® MKL)
3. Intel® C++ Compiler with Intel® Cilk™ Plus

These components can now target embedded systems. Intel System Studio has:

- An extended OS support matrix including custom images built with the Yocto Project™
- Cross-compilation support
- Numerous analysis features (remote, power, memory, SoC)
- Low-level debug capabilities in addition to application-level debugging.

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).

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 { // 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).

¹ Usually highly optimized library code is reused, whereas custom code is only developed when necessary.
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 Z (Intel® AVX 2).

#### 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.

#### Pipeline Pattern

```cpp
// Start using Intel IPP!
ippInit();

int main(int argc, char* argv[]) {
  // Example driver along with an application requesting this driver's service.
  // Such an example may help to understand the best application path. A dynamic link library (*.so, *.dll) allows for an initialization step by capturing the "load event" of the code path. A dynamic link library (*.so, *.dll) allows for an initialization step by capturing the "load event" of the code path.
  ...
}
```

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" for single-precision floating point). C language call-convention and linkage

![Image 4](https://example.com/image4.png)

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.

The parallelization level decision is a trade-off between high 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).

```
def read_signal(size, x, y, stage):
    return read_signal(16, x, y, stage)
```

Figure 3: The noise function scales normalized random values of the interval [-1,+1] by a certain level and adds it to the result of a unary function. This unary function is given via function pointer.

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

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 (3) reads its own buffer #4 and prints to standard output.

```
struct { size_t i, n; } stage[] = {{ 0, 0 }, { 1, 0 }, { 2, 0 }, { 3, 0 }};

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, stage[1].n);
    print_signal (stage[2].n, x + stage[2].i, y + stage[2].i);
    stage[2].n = stage[1].n;
    std::rotate(stage, stage + 4 - 1, stage + 4); // quad-buffering
}
```

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.

### Intel® IPP Functions

- **Transforms (e.g., rotation)**
- **Convolution / Correlation**
- **Filtering (e.g., IIR, FIR)**
- **Statistics**
- **Trigonometric functions**
- **Decomposition, Eigenvalues**
- **Logical, shift, conversion**
- **Error correction, Reed-Solomon**
- **Compression (entropy, dict.)**
- **FFT, DFT, DCT**
- **Integrity / Compression / Cryptography**
- **Vector / Matrix**
- **Color Conversion**
- **Utilities**

#### More Domains...

- **Video, picture coding**
- **Audio (e.g., speech coding)**
- **String processing**
- **Resampling**

#### Intel® IPP Capabilities

- **Streaming**: Intel IPP provides internal multithreading based on OpenMP®.

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

**License Information**

Copyright © 2013, Intel Corporation. All rights reserved.

For more information about Intel IPP, please visit: [Intel IPP](https://software.intel.com/en-us/ipp)
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 multicores. The Intel Compiler enables auto-vectorization and targets at least the SSE2 instruction set extension by default.\(^5\) 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) for effective auto-vectorization.

\(^5\) The 64-bit application binary interface (ABI) describes SSE2 as mandatory.
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.

```
struct background {
  // proper shutdown
  int i;
  std::thread thread;  // run a task in the background
};
```

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_CPP11). Note with Intel System Studio, Intel TBB can be built from source.

Intel® Cilk™ Plus

```
for (size_t i = 0; i < N; ++i) {
  cilk_spawn (size_t i = 0; i < N; ++i) {
    cilk_for (size_t I, size_t N) {
      /*A*/
      work(size_t i, size_t N){
        for (size_t i = 0; i < N; ++i) {
          /*B*/
          cilk_for (size_t I, size_t N) {
            cilk_spawn (size_t i = 0; i < N; ++i) {
              cilk_sync;
            }
          }
        }
      }
    }
  }
}
```

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).

```
for (size_t i = 0; i <= nsteps && (0 < stage[1].n || 0 == i); ++i) {
  cilk_spawn (size_t i = 0; i <= nsteps && (0 < stage[1].n || 0 == i); ++i) {
    cilk_for (size_t I, size_t N) {
      cilk_spawn (size_t i = 0; i <= nsteps && (0 < stage[1].n || 0 == i); ++i) {
        cilk_sync;
      }
    }
  }
}
```

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 15) 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
Intel Compiler’s auto-vectorization is quite advanced and given it is effective; one would need to The array notation offers a nice syntax to control the way how code is vectorized. This can be very provides a well-optimized function to perform the FIR filter.

outer loop is to harvest parallelism for multiple cores and where the inner loop is vectorized (SIMD). Note Intel IPP

![Figure 18](image1.png) Here, the inner loop now uses Intel Cilk Plus’ Array notation. This is a typical implementation where the

increasing the grain size lowers the number of worker involved into a cilk_for loop. The default (when the pragma is not used) aims to maximize parallelism, and involves (1) the size of the iteration space as well as (2) the total number of workers (see Figure 16). A constant grain size 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 calculations impacts the final result. Other reasons are the code path taken, or the data alignment (loop prolog/remainder). There are actually many more reasons for not reproducing the same bit-wise result on a system or across systems.

Array Notation: Intel Cilk Plus supports an extended array notation and vectorized elemental functions in addition to auto-vectorization, and in addition to ISA-specific intrinsic functions.

![Figure 17](image2.png) One can find a fragment of the above example (Finite Impulse Response / FIR filter) also within the

"Intel C++ Compiler User and Reference Guide". However, cilk_for has been applied to the outer loop of the

parallelizing the signal processing algorithm itself. Instead, the pipeline pattern (often applicable for

"kernels" are the only way to express parallelism, a not so obvious synchronization might be

manualy close the gap to the heuristics that are applied during auto-vectorization. Array notation leads to elegant and readable code that directly models data-parallelism using vector processing.2

![Figure 19](image3.png) Here, the inner loop of the initial implementation is interleaved 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).

![Figure 20](image4.png) 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 ("__declspec" or "__attribute__") along with optional clauses ("uniform") that allows to compile this kernel in a separate translation unit but to guarantee vectorization for the call-side. This is quite an important property since loop vectorization usually requires seeing the entire loop body i.e., to inline the code. Here a "vector function" compiles, and may end up in multiple vectorized loops without code bloat. Note advanced techniques such as link-time code generation across multiple translation units (i.e., called IPO by the Intel Compiler) are not necessary in this case.

An elemental function is a "kernel" in a narrow sense. With respect to the parallel pattern, a kernel itself does not carry parallelism inside. Instead, a kernel is launched element-wise over the problem domain. This usually implies being unable to scatter into neighboring positions. In cases where "kernels" are the only way to express parallelism, a not so obvious synchronization might be employed by grouping teams of threads to exchange data locally. However, a signal processing filter can be often expressed as a kernel function that launches over a linear range. Array notation with elemental functions allows expressing this directly including a convenient syntax to launch the kernel function over a range (see Figure 19). A linear range is equivalent to a single loop whereas an imaging filter that is launched over a 2d region would correspond to two nested loops, etc.

**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.

---

[2] 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
Our Intel Core i5-2640M based (dual core) system achieved a speedup of more than 1.6x for
the signal into a file. 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).

The Intel Cilk Plus based executable with one worker was exactly on par with the single-threaded
 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 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 multcore 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.

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.

Softwaroptimizations 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 cores².

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).

7 This implies to perfectly parallelize an application.