Threading Intel® Integrated Performance Primitives Image Resize with Intel® Threading Building Blocks

The Intel® Integrated Performance Primitives (Intel® IPP) library provides a wide variety of vectorized signal and image processing functions. Intel® Threading Building Blocks (Intel® TBB) adds simple but powerful abstractions for expressing parallelism in C++ programs. This article presents a starting point for using these tools together to combine the benefits of vectorization and threading to resize images.

From Intel® IPP 8.2 onwards multi-threading (internal threaded) libraries are deprecated due to issues with performance and interoperability with other threading models, but made available for legacy applications. However, multithreaded programming is now main stream and there is a rich ecosystem of threading tools such as Intel® TBB. In most cases, handling threading at an application level (that is, external/above the primitives) offers many advantages. Many applications already have their own threading model, and application level/external threading gives developers the greatest level of flexibility and control. With a little extra effort to add threading to applications it is possible to meet or exceed internal threading performance, and this opens the door to more advanced optimization techniques such as reusing local cache data for multiple operations. This is the main reason to start deprecating internal threading in the latest releases.

Getting started with parallel_for

Intel® TBB's parallel_for offers an easy way to get started with parallelism, and it is one of the most commonly used parts of Intel® TBB. Any for() loop in the applications, where each iteration can be done independently and the order of execution doesn't matter. In these scenarios, Intel® TBB parallel_for is useful and takes care of most details, like setting up a thread pool and a scheduler. You supply the partitioning scheme and the code to run on separate threads or cores. More sophisticated approaches are possible. However, the goal of this article and sample code is to provide a simple starting point and not the best possible threading configuration for every situation.

Intel® TBB's parallel_for takes 2 or 3 arguments.

parallel_for ( range, body, optional_partitioner )

The range, for this simplified line-based partitioning, is specified by:
This provides information to each thread about which lines of the image it is processing. It will automatically partition a range from begin to end in grainsize chunks. For Intel® TBB the grainsize is automatically adjusted when ranges don't partition evenly, so it is easy to accommodate arbitrary sizes.

The body is the section of code to be parallelized. This can be implemented separately (including as part of a class); though for simple cases it is often convenient to use a lambda expression. With the lambda approach the entire function body is part of the parallel_for call. Variables to pass to this anonymous function are listed in brackets \[alg, pSrc, pDst, stridesrc_8u, \ldots\] and range information is passed via blocked_range<int>& range.

This is a general threading abstraction which can be applied to a wide variety of problems. There are many examples elsewhere showing parallel_for with simple loops such as array operations. Tailoring for resize follows the same pattern.

**External Parallelization for Intel® IPP Resize**

A threaded resize can be split into tiles of any shape. However, it is convenient to use groups of rows where the tiles are the width of the image.

Each thread can query range.begin(), range.size(), etc. to determine offsets into the image buffer. Note: this starting point implementation assumes that the entire image is available within a single buffer in memory.

The new image resize functions in Intel® IPP 7.1 and later versions, new approach has many advantages like

- IppiResizeSpec holds precalculated coefficients based on input/output resolution combination. Multiple resizes which can be completed without recomputing them.
- Separate functions for each interpolation method.
- Significantly smaller executable size footprint with static linking.
- Improved support for threading and tiled image processing.
- For more information please refer to article: [Resize Changes in Intel® IPP 7.1](#)

Before starting resize, the offsets (number of bytes to add to the source and destination pointers to calculate where each thread's region starts) must be calculated. Intel® IPP provides a convenient function for this purpose:

```cpp
ippiResizeGetSrcOffset
```
This function calculates the corresponding offset/location in the source image for a location in the destination image. In this case, the destination offset is the beginning of the thread's blocked range.

After this function it is easy to calculate the source and destination addresses for each thread's current work unit:

\[
\begin{align*}
  pSrcT &= pSrc + (srcOffset.y \times stridesrc\_8u) \\
  pDstT &= pDst + (dstOffset.y \times stridedst\_8u)
\end{align*}
\]

These are plugged into the resize function, like this:

```c
ippiResizeLanczos\_8u\_C1R(pSrcT, stridesrc\_8u, pDstT, stridedst\_8u, dstOffset, dstSizeT, ippBorderRepl, 0, pSpec, localBuffer);
```

This specifies how each thread works on a subset of lines of the image. Instead of using the beginning of the source and destination buffers, \( pSrcT \) and \( pDstT \) provide the starting points of the regions each thread is working with. The height of each thread's region is passed to resize via \( dstSizeT \). Of course, in the special case of 1 thread these values are the same as for a nonthreaded implementation.

Another difference to call out is that since each thread is doing its own resize simultaneously the same working buffer cannot be used for all threads. For simplicity the working buffer is allocated within the lambda function with scalable_aligned_malloc, though further efficiency could be gained by pre-allocating a buffer for each thread.

The following code snippet demonstrates how to set up resize within a `parallel_for` lambda function, and how the concepts described above could be implemented together.

[Click here](#) for full source code.

By downloading this sample code, you accept the [End User License Agreement](#).

As you can see, a threaded implementation can be quite similar to single threaded. The main difference is simply that the image is partitioned by Intel® TBB to work across several threads, and each thread is responsible for groups of image lines. This is a relatively straightforward way to divide the task of resizing an image across multiple cores or threads.

**Conclusion**

Intel® IPP provides a suite of SIMD-optimized functions. Intel® TBB provides a simple but powerful way to handle threading in Intel® IPP applications. Using them together allows access to great vectorized performance on each core as well as efficient partitioning to multiple cores.
The deeper level of control available with external threading enables more efficient processing and better performance.

For more complete information about compiler optimizations, see our Optimization Notice.