A new article about using Intel TBB is here . It contains examples using C++ lambdas and joining multi-threaded loops with SIMD code

In this article we will transform a plain C loop into a multi-threaded version using Intel Thread Building Blocks library (TBB).

Here is the loop to transform:

{CODE brush: cpp; ruler: true;}

```
 unsigned char *SrcImagePtr = (unsigned char *)SrcImage;
 unsigned char *DstImagePtr = (unsigned char *)DstBuffer;
 for (int i = (OriginalImageWidth * OriginalImageHeight); i > 0; i--)
 {
 int YValue = (SrcImagePtr[0] * FirstFactor ) +
 (SrcImagePtr[1] * SecondFactor) +
 (SrcImagePtr[2] * ThirdFactor );
 SrcImagePtr += PixelOffset;
 YValue += 1 << (SCALING_LOG - 1);
 YValue >>= SCALING_LOG;
 if (YValue > 255)
 YValue = 255;
 *DstImagePtr = (unsigned char)YValue;
DstImagePtr++;
```

{/CODE}

This loops iterates over a three-channel image named SrcImage (usually a RGB one), and it computes the luma value for each pixel storing it into DstImage. As the computation of every pixel has no dependencies whatsoever on other pixel, it is very simple to separate this computation into multiple threads, each performing it on a different slice of the image.

Even if we could directly use threads for such a task, it is much simpler and faster to use an ad-hoc library such as Intel's Thread Building Blocks.

The first step is including the relevant header files:

{CODE brush: cpp; ruler: true;} #include <task_scheduler_init.h>
using namespace tbb;
#include <parallel_for.h>
#include <blocked_range.h>
#include <scalable_allocator.h> {/CODE}

Including scalable_allocator is not mandatory, but if any of the threads allocates any memory, switching to a memory manager that performs better with multi-threaded memory request avoids a potential performance bottleneck. After that, we make sure that both the include and library paths of TBB are included in the project, and that TBB DLL libraries are in the application path.

Before writing multi-threaded loops, we must initialize the TBB task scheduler by creating a task_scheduler_init instance. We can initialize it in the startup phase of the software app, but it is more reliable to just build a class to do it:

{CODE brush: cpp; ruler: true;} class CTBBInit
{
public:
 CTBBInit()
 {
 /// init Intel TBB
 task_scheduler_init init;
 }
 ~CTBBInit()
 {

 }
} TBBInit; {/CODE}

We have finished the preliminary steps, now we can write the code that actually does the computation. For every loop that we transform into a multi-threaded one, we must declare a class like the following one:

{CODE brush: cpp; ruler: true;}struct GenericToGrayConverter {
unsigned char *SrcImagePtr;
unsigned char *DstImagePtr;
int PixelOffset;
int FirstFactor;
int SecondFactor;
int ThirdFactor;
void operator( )(const blocked_range<int>& range) const {
unsigned char *LocalSrcImagePtr = SrcImagePtr;
unsigned char *LocalDstImagePtr = DstImagePtr;
LocalSrcImagePtr += range.begin() * PixelOffset;
LocalDstImagePtr += range.begin();
for(int i=range.begin(); i!=range.end( ); ++i)
{
int YValue = (LocalSrcImagePtr[0] * FirstFactor) +
(LocalSrcImagePtr[1] * SecondFactor) +
(LocalSrcImagePtr[2] * ThirdFactor);
LocalSrcImagePtr += PixelOffset;
YValue += 1 << (SCALING_LOG - 1);
YValue >>= SCALING_LOG;
if (YValue > 255)
YValue = 255;
*LocalDstImagePtr = (unsigned char)YValue;
LocalDstImagePtr++;
}
}
}; {/CODE}

The variables of the class are a copy of the data used in the loops (they will be initialized by the calling routine, as we will see later). Then we redefine the operator () to perform the given computation on a slice of the original data set, delimited by range.begin() and range.end(), so the image pointers are adjusted by adding range.begin() data elements before starting the loop, and the loop index goes from range.begin() to range.end(). Please note that we cannot modify the class variables like SrcImagePtr and DstImagePtr inside the operator (), so we define a local copy of them in the operator body (named LocalSrcImagePtr and LocalDstImagePtr) and work with them.

The final step is replacing the original loop with a new code fragment that starts the working threads:

{CODE brush: cpp; ruler: true;}GenericToGrayConverter GenericToGrayConverterPtr;
GenericToGrayConverterPtr.FirstFactor = FirstFactor;
GenericToGrayConverterPtr.SecondFactor = SecondFactor;
GenericToGrayConverterPtr.ThirdFactor = ThirdFactor;
GenericToGrayConverterPtr.PixelOffset = PixelOffset;
GenericToGrayConverterPtr.ImageBufferPtr = ImageBufferPtr;
GenericToGrayConverterPtr.ImagePtr = ImagePtr;
parallel_for(blocked_range<int>(0, (OriginalImageWidth * OriginalImageHeight), (OriginalImageWidth * OriginalImageHeight) >> 3), GenericToGrayConverterPtr);{/CODE}

In this code fragment, we first create an instance of the class we just defined (GenericToGrayConverter), then we init the class variables so that they contain a copy of the data needed by the computation, finally we start the computation with a parallel_for statement. Please note that the third parameter of the parallel_for statement contains the size of each slice of the image that will be computed by a thread, in this example the image was sliced in eight parts so that each part is large enough to make threading overhead negligible, still there's enough level of parallelism to fully use up to 8 cores.

Summing up, converting a serial loop to a multi-threaded one may look complex at the beginning, but thanks to libraries such as Intel's TBB, it can be done with a minimal amount of code, and the speed-up that can be achieved on current 4- or 8-cores processors greatly justifies the development time.