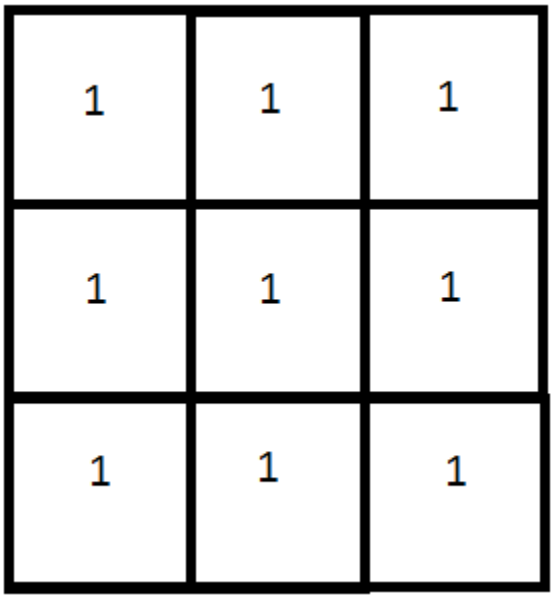
Improving Averaging Filter performance using Intel® Cilk™ Plus

Introduction

Intel® Cilk™ Plus is an extension to the C and C++ languages to support data and task parallelism. It provides three new keywords to implement task parallelism and Array Notation, simd pragma and Elemental Function to express data parallelism. This article demonstrates how to improve the performance of an Averaging Filter in image processing using Intel® Cilk™ Plus. To demonstrate the performance increase, you will use a program that reads a bitmap RGB image and does averaging with a filter of size 3x3. Averaging filter works by averaging the pixel values found in the immediate neighborhood of the current pixel. The immediate neighborhood means neighbors in all 8 directions which are 1 unit distance away from the current pixel.

Overview

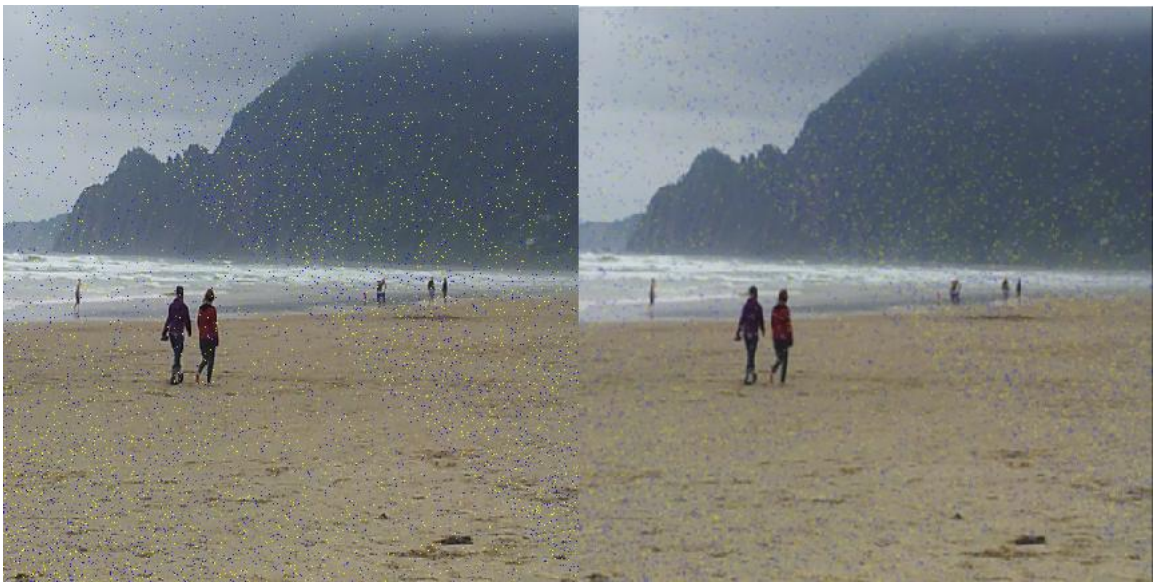
An Averaging filter is a commonly used filter in the field of image processing and is mainly used for removing any noise in a given image. A noise in an image is any presence of pixel values which doesn't blend well with the actual content of photo like salt-pepper grains on an image. Averaging filter makes use of the concept that any given pixel value will not change drastically from its immediate neighbors. In other words, the current value of a pixel depends more on its immediate neighbors. Averaging filter size decides how many immediate neighbors are considered for the computation of the current pixel values. Most commonly used filter size 3x3. The filter is two dimensional since an image nothing but a 2D signal.



The figure above gives a visual depiction of how an averaging filter of size 3x3 looks. The 1s in each box represents the weights for the immediate neighbor's pixel values. Averaging filter operation on an image is a general 2D convolution problem. The filter is taken and placed such that the middle box of the filter sits on every pixel of the original image and the computation is done as shown mathematically below:

$$Value\ of\ current\ pixel = \sum_0^2 \sum_0^2 (filter(i,j) * image(i,j))$$

The image which the program works with is in RGB bitmap format. R, G, and B are the Red, Green, and Blue values of each pixel in the input image. Though every pixel value is determined by its neighbors, still this is a highly data parallel algorithm considering the fact that multiple pixel values can be computed at the same time using the SIMD (Single Instruction Multiple Data) feature where multiple data items in a loop construct are loaded into the vector registers and operated upon simultaneously by a single instruction. Below is the bitmap file before and after applying Averaging Filter:



a) Original Image

b) Filtered Image

a) Original Image b) Filtered Image

We will look at the performance of the serial implementation of the Averaging filter algorithm and then create an Intel® Cilk™ Plus implementation of the filter to improve filter's performance through vectorization and parallelization features supported by Intel® C++ Compilers.

Optimization Steps

We will start the optimization process by performing the following steps:

- Establish a performance baseline by building and running the serial version of the Averaging filter with Intel® C++ Compiler (Release Build).
- Rewrite the serial implementation by interpreting the same data structure in a different way without changing the data layout in memory and see the improvement in performance of the new serial version (Release Build).
- Implement the filter using Intel® Cilk™ Plus Array Notation considering Array of Structure layout of image.
- Replace the Array of Structure (AOS) implementation with Structure of Array (SOA) implementation to improve performance further.
- Introduce threading using cilk_for on higher resolution images to improve the performance further by using multiple cores.

System Requirements

To compile and run the example and exercises in this document you will need Intel® C++ Composer XE 2013 Update 1 or higher, and an Intel® Pentium 4 Processor or higher with support for Intel® SSE2 or higher instruction extensions. **The exercises in this document were tested on a third generation Intel® Core™ i5 system supporting 256-bit vector registers.** The instructions in this document show you how to build and run the examples with the Microsoft Visual Studio* 2012. A Visual Studio* 2008 project is provided to allow using the examples with older versions of Visual Studio*. The examples provided can also be built from the command line on Windows*, Linux*, and Mac OS* X using the following command line options:

```
Windows*: icl /Qvec-report2 /Qrestrict /fp:fast
AveragingFilter.cpp

Linux* and Mac OS* X: icc -vec-report2 -restrict -fp-model -fast
AveragingFilter.cpp
```

For system requirements for Linux and Mac OS X please refer to the Intel® C++ Composer XE 2013 Release Notes.

NOTE: The sample code used in this article will only read images in RGB (24 bit format) and with .bmp extensions. Two sample images are attached with this solution.

Locating the Samples

To build the sample code, open the AveragingFilter.zip archive attached.
Use these files for this tutorial:

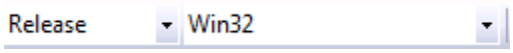
There are sample input images (lenna.bmp and blackbuck.bmp) in the AveragingFilter directory inside the zip file.

- `silver512.bmp`
- `nahelam512.bmp`

The above sample input images are in the AveragingFilter directory inside the zip file.

- `AveragingFilter.sln`
- `AveragingFilter.cpp`
- `AveragingFilter.h`
- `Timer.h`
- `Timer.cpp`
- `Open the Microsoft Visual Studio* solution file, AveragingFilter.sln,` and follow the steps below to prepare the project for the exercises in this document:

1. Select "Release" "Win32" configuration



2. Clean the solution by selecting **Build > Clean Solution**.

You just deleted all of the compiled and temporary files association with this solution. Cleaning a solution ensures that the next build is a full build rather than changing existing files.

Contents of the Source Code

The program has a main function that gets the input file and output file as command line arguments and invokes the read_process_write() function. This function does the reading of the .bmp input file. The function first reads the header information from the input image file which contains information about the type of image, compression if any, and width and height of the input image. Once this information is known, a dynamic data structure is created and the payload image data is copied on to this data structure for further processing at pixel level. In this program, both array of structure (AOS) and structure of array (SOA) versions of the data structures are implemented and their performance is compared.

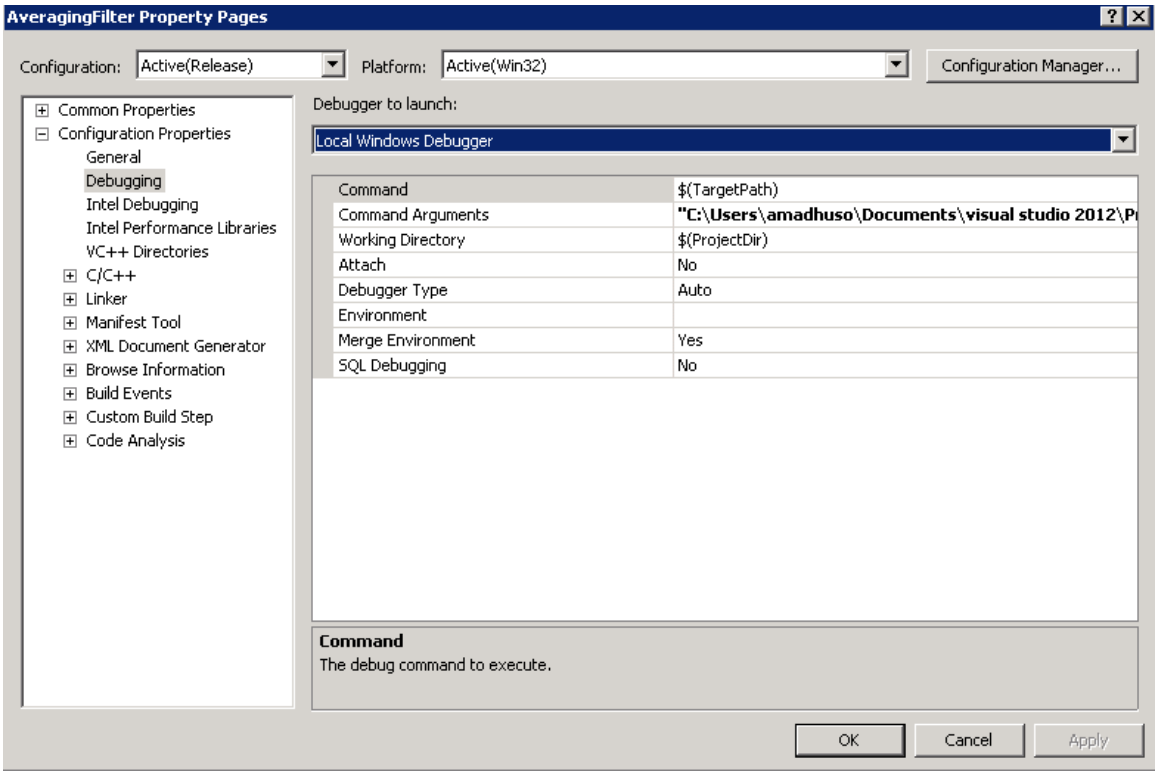
The main Averaging filter kernel is named process_image() and depending on the macro used during the compilation phase, the corresponding implementation of the Averaging kernel is enabled (for instance array notation version or cilk_for version implemented using SOA and AOS Data structures).

To run the executable from the command line, please use the following command line options:

<executable> <input file> <output file>

The input image and output image can be fed as command line arguments in Visual Studio as follows:

Right Click on the project > Properties > Configuration Properties > Debugging > Command Arguments



Establishing a Performance Baseline

To set a performance baseline for the improvements that follow in this tutorial, build your project with the Intel® C++ compiler in Visual Studio* (on Windows*). Run the executable (**Debug > Start without Debugging**). Also Use AVX2 instruction set if you have a Haswell processor else go for AVX instruction set. Selection of the instruction set can done by **Right Clicking project > Configuration properties > C/C++ > Code Generation[Intel C++] > Intel Procesor-Specific Optimization > Select SSE/AVX/AVX2**. Running the program results in starting a window that displays the program's execution time in number of clock ticks. Record the execution time reported in the output. It is advised to use AVX2 if you have Haswell processor because AVX2 instructions support 256 bit integers SIMD operations unlike the AVX instructions which supports 256 bit SIMD operations only on floats and doubles. Here the data type used is unsigned short and thus better utilization of SIMD registers (YMM register) can be ensured by using AVX2 instructions.

Rewriting the Serial Kernel in a different way for better performance

The previous implementation of averaging filter kernel involved writing the logic of averaging filter with structure RGB and accessing the individual red, green and blue values of RGB structure with dot operator.

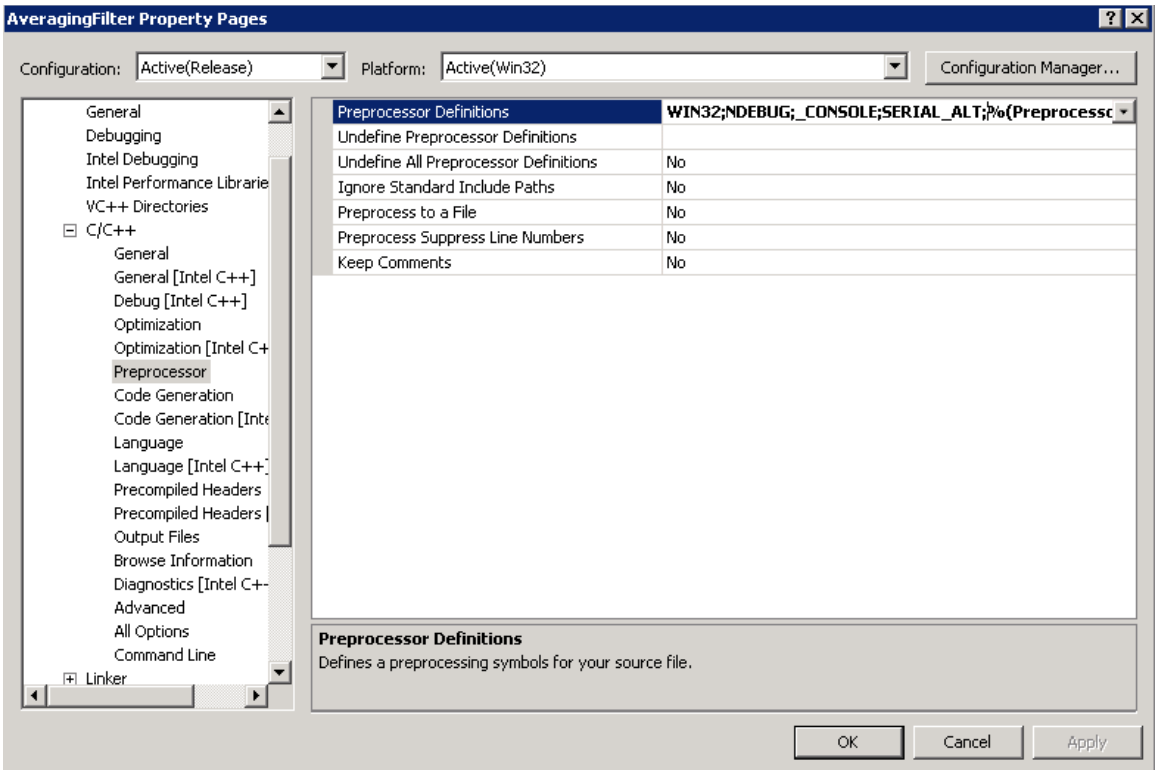
```
            for(int k1 = (-1); k1 <= 1; k1++)
            {
                int pos = j + (k1 * resized_width);
                for(int k2 = (-1); k2 <= 1; k2++)
                {
                    resized_outdataset[j].red += resized_indataset[(pos + k2)].red;
                    resized_outdataset[j].green += resized_indataset[(pos + k2)].green;
                    resized_outdataset[j].blue += resized_indataset[(pos + k2)].blue;
                }
            }
            resized_outdataset[j].red /= 9;
            resized_outdataset[j].green /= 9;
            resized_outdataset[j].blue /= 9;
```

The compiler generated vectorization report states the following for the above code snippet:

```
AveragingFilter.cpp(131): message : loop was not vectorized: existence of vector
dependence
AveragingFilter.cpp(128): message : loop was not vectorized: not inner loop
AveragingFilter.cpp(126): message : loop was not vectorized: not inner loop
AveragingFilter.cpp(123): message : loop was not vectorized: not inner loop
```

Though the above loop has potential data parallelism, the compiler didn't have enough information given the way the code is written. Hence the same serial kernel is rewritten but this time with a different perspective of the AOS data structure which holds the payload of the image. Though the payload is stored in a Array of Structures each structure consist of 3 unsigned shorts (red, green and blue). So eventually this array of structure can also been interpreted as a single array of unsigned short. This interpretation perfectly works because the array of structures are also laid out in memory in a similar fashion as the array unsigned shorts in this case.

With this basic change in perspective, the serial kernel is rewritten (Code block which gets enabled by enabling macro "**SERIAL_ALT**" in the code sample)



and the vectorization report compiler generates this time is:

```
AveragingFilter.cpp(97): message : LOOP WAS VECTORIZED
```

So all it takes is to express the logic in a more compiler friendly way and compiler will generate a better performing code by making use of the underlying SIMD units. This executable performs the job faster than the previous serial version. The only drawback with this implementation is that we need three different expressions, each for computing the filtered value of red, green and blue pixels. In the next section you can compute the result of red, green and blue pixels in the same statement by using vector operands rather than using traditional scalar operands. Intel® Cilk™ Plus Array Notation can be used to explicitly specify the vector operands.

Implementation of Sepia filter kernel using Array Notation

Here we re-write the previous kernel using the Array Notation. In order to build the Array Notation version, Select Project **> Properties > C/C++ > Preprocessor > Preprocessor Definitions**, and add a new macro "**ARRAY_NOTATION**".

1. Rebuild the project, then run the executable (**Debug > Start Without Debugging**) and record the execution time reported in the output. Array notations version will make use of the SIMD registers and SIMD instruction set to handle operations on vector operands. The vectorization report shows that the array notation version of the loop got vectorized:

```
AveragingFilter.cpp(117): message : LOOP WAS VECTORIZED
```

For our Averaging filter example the performance result of the Array Notation implementation will be better than the previous implementations. The benefit of using Array Notation is that while vectorizing arbiterary code is at the discretion of the compiler and cannot always be guaranteed, using array notation guarantees vectorization. As you notice with every implementation, the utilization of single core SIMD units are increasing.

Improving Performance Further using Structure of Arrays (SOA)

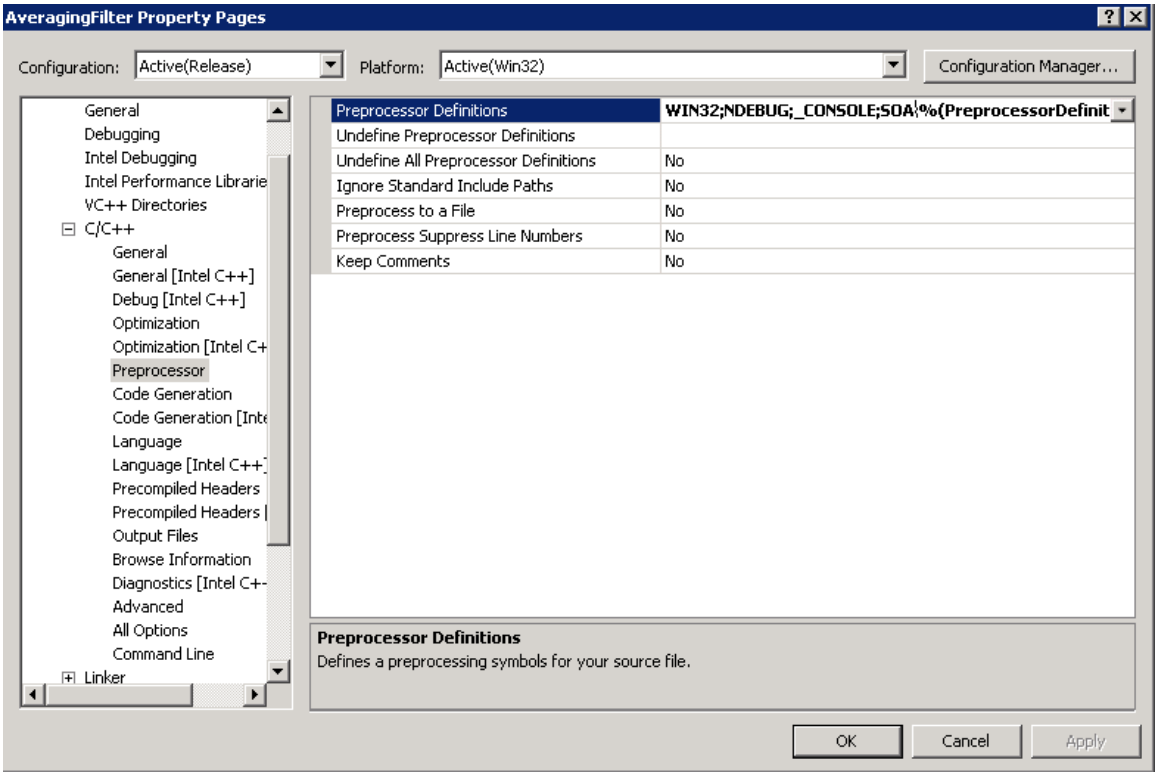
Up until now our default implementation has been using an Array of Structure algorithm that is not very vectorization friendly due to non-sequential access patterns. The non-sequential access pattern results in generating gather/scatter instructions that reduce the vectorization efficiency due to the long instruction latencies. Despite the above implications, Intel® Cilk™ Plus was able to give admirable performance. By rewriting the baseline implementation in Structure of Array (SOA) we could further improve performance due to unit-stride memory access pattern which is vectorization friendly. This allows the compiler to generate faster linear vector memory load/store instructions (e.g. movaps or movups supported on Intel® SIMD hardware) rather than generating longer latency gather/scatter instructions that it would have to do otherwise. The data structure used in Array of Structures (AOS) implementation is:

```
typedef struct {
    unsigned char blue;
    unsigned char green;
    unsigned char red;
} rgb;
```

For Structure of Arrays (SOA) implementation, the data structure used is as follows:

```
typedef struct {
    unsigned char *blue;
    unsigned char *green;
    unsigned char *red;
} SOA_rgb;
```

In order to enable this section of the code in the example simply enable the macro "SOA" as shown below:



Rebuild the project with the above setting to vectorize the code:

```
AveragingFilter.cpp(62): message : PARTIAL LOOP WAS VECTORIZED
AveragingFilter.cpp(62): message : PARTIAL LOOP WAS VECTORIZED
AveragingFilter.cpp(62): message : PARTIAL LOOP WAS VECTORIZED
```

The function process_image() containing the SOA implementation doesn't use any Intel® Cilk™ Plus features in specific but here the auto-vectorizer does a great job in vectorizing the code.

The performance number should show a significant improvement over its AOS counterpart earlier without using Array Notations. But as noticed before, it is always good to write a code which always guarantees vectorization. Using Array notation or simd pragma with relevant clauses will guarantee vectorization across different versions of Intel® C++ Compiler.

There is also a corresponding array notation implementation of the logic in SOA and that section of code can enabled using macros "**SOA_AN**" and "**ARRAY_NOTATION**". This implementation demonstrates the explicit vectorization using array notation to express that logic to the compiler.

Improving Performance Further using cilk_for

Till now the focus was to make use of SIMD units in a single core. Now that good utilization of single core is ensured, the same logic can scaled across multiple cores using cilk_for which depending on the availability of cilk worker threads will

parallelize across multiple cores. To try this, you should make sure the workload passed to the program is good enough for parallel processing. The sample code attached has a couple high resolution images (3264x2448).

You can change the "for" to "cilk_for" at line number 45 to make sure threading solution will be used depending on the availability of the cilk worker threads. By default Cilk runtime creates as many worker threads as the number of physical cores on the machine. When a cilk_for block is encountered, if there is any available cilk worker thread, the total iteration space is divided into multiple mutually exclusive parts. This process will make sure that image is divided and processed on multiple cores. If the workload (the size of the image), is small like 512x512, the spawning overhead of cilk worker threads will start to show up significant. So when using threading solution it is good make sure the workload is big enough.

References

For more information on SIMD vectorization, Intel® C++ Compiler automatic vectorization, Elemental Functions and examples of using other Intel® Cilk™ Plus constructs refer to:

"A Guide to Autovectorization Using the Intel® C++ Compilers"
"Requirements for Vectorizing Loops"
"Requirements for Vectorizing Loops with #pragma SIMD"
"Getting Started with Intel® Cilk™ Plus Array Notations"
"SIMD Parallelism using Array Notation"
"Intel® Cilk™ Plus Language Extension Specification"
"Elemental functions: Writing data parallel code in C/C++ using Intel® Cilk™ Plus"
"Using Intel® Cilk™ Plus to Achieve Data and Thread Parallelism"