TBB: parallel_for

OBJECTIVES

- Learn the basics of Threading Building Blocks (TBB) Library in C++: *parallel_for*
- Execute parallel applications using TBB.

USEFUL INFORMATION

- Refer to the Tutorial: Embedded Intel for the source files used in this Tutorial.
- Refer to the board website or the Tutorial: Embedded Intel for User Manuals and Guides for the Terasic DE2i-150 Board.
- Board Setup: Connect the monitor (VGA or HDMI) as well as the keyboard and mouse.
  - ✓ Refer to the *DE2i-150 Quick Start Guide* (page 2) for a useful illustration.

ACTIVITIES

FIRST ACTIVITY: ELEMENT-WISE SQUARING OPERATION IN A VECTOR

- Given an n-element vector, we want to compute $a(i) \leftarrow a(i) \times a(i)$ for every element in the vector.
- Example: `a = [-1.2 2.3 3.6 6.7 0.3 0.35 2.1 0.7 5.1 -1.1]`
  Result: `a = [1.44 5.29 12.96 44.89 0.09 0.1225 4.41 0.49 26.01 1.21]`

- This simple example will introduce the concept of the *parallel_for* template function in TBB. This template function can be used quickly when the iterations are independent.

Sequential implementation

- This is a straightforward sequential implementation as a function:

```
void SerialApplyFun( float a[], size_t n ) {
    for( size_t i=0; i!=n; ++i )
        a[i] = a[i]*a[i];
}
```

Parallel implementation

- TBB lets us specify logical parallelism instead of threads.
  - ✓ Programming directly in terms of threads can be tedious and lead to suboptimal programs (threads are low-level, heavy constructs close to the hardware). It also forces you to efficiently map logical tasks onto threads.
  - ✓ In contrast, TBB library automatically maps logical parallelism in a way that makes efficient use of processor resources.
- The functionality of the `SerialApplyFun` function can be implemented (with parallelism enabled) via Intel TBB *parallel_for*. The iterations are divided up into tasks that are provided to a task scheduler for parallel execution.
  - ✓ Style 1: Using a class.
    - ▫ The *parallel_for* template indicates the iteration space (`blocked_range`: half-open range from `0` to `n-1`), as well as the function name and operand to which it applies (`ApplyFun(a)`):

```
void ParallelApplyFun( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0,n), ApplyFun(a) );
}
```

    - ▫ `ApplyFun(a)`: It applies $a(i) \leftarrow a(i) \times a(i)$ for `i = 0` to `n-1`. It must be defined as a *class* in a specific way:

```
class ApplyFun {
    float *const my_a; // 'private' access (default access level)
public:
    void operator()( const blocked_range<size_t> &r ) const {
        float *a = my_a;
        for( size_t i=r.begin(); i!=r.end(); ++i ) // you can use 'size_t', long int or int
           a[i] = a[i]*a[i];
    }
    ApplyFun( float a[] ) : my_a(a) {} // member initialization in constructor. my_a = a
};
```

    - ▫ Because the body object might be copied, its `operator()` should not modify the body. Otherwise the modification might or might not become visible to the thread that invoked *parallel_for*, depending upon whether `operator()` is acting on the original or a copy. As a reminder of this nuance, *parallel_for* requires that the body object's `operator()` be declared `const`.

- Style 2: Using a structure.
  - The *parallel_for* template indicates the iteration space (`0` to `n-1`), and the function name to which it applies: `ApplyFun`. However, the function parameters (`a` in this case) must be provided in a structure called `ApplyFun`.

```
void ParallelApplyFun( float a[], size_t n ) {
   ApplyFun.a = a;
   parallel_for(blocked_range<size_t>(0,n), ApplyFun);
}
```

  - `ApplyFun`: It applies $a(i) \leftarrow a(i) \times a(i)$ for `i = 0` to `n-1`. It must be defined as a structure that contains the function definition and the operator:

```
struct ApplyFun {
    float *a;
    void operator()( const blocked_range<size_t> &r ) const {
        for( size_t i=r.begin(); i!=r.end(); ++i )
           a[i] = a[i]*a[i];
    }
};
```

- Note that *parallel_for(range, body)* represents parallel execution of body over each value in the range.
- Fig. 1 depicts the sequential and parallel implementation of this operation for $n=4$. Ideally, all the operations with *parallel_for* run in parallel (after an overhead).

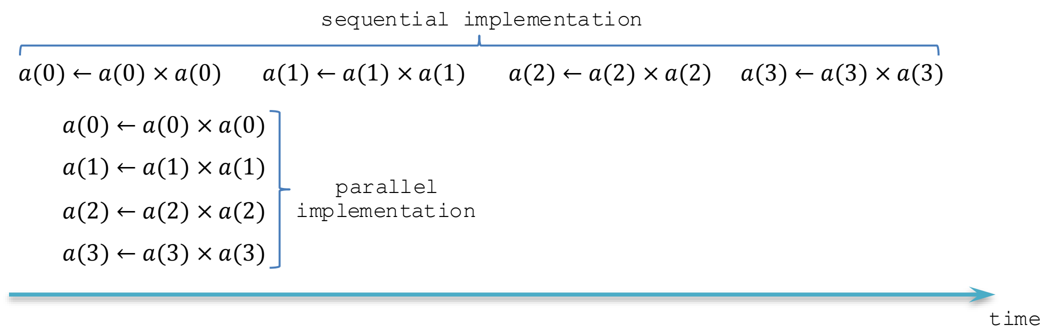

Figure 1. Timeline: Sequential vs parallel implementation for the operation applied on a 4-element vector

- `blocked_range` template class: specification of discrete ranges with enumerable elements.
  - `blocked_range<type>(a,b)`: It represents a half-open range `[a,b)` that can be recursively split.
    Example: `blocked_range<int>(0,5)`: The range is `[0,5)` and it includes the numbers 0,1,2,3,4.
  - `blocked_range<type>(a,b, grainsize)`: It represents a half-open range `[a,b)` with a grainsize (which is 1 by default).
    Example: `blocked_range<int>(5,14,2)`: The range is `[0,14)` but it only includes the numbers 5, 7, 9, 13.
  - `blocked_range<type> &r`: It defines `r` as a blocked range of 'type'.

- Application file: `basic.cpp`
  - In this example, `a` is a constant vector with `n = 10` elements.
  - The operation $a(i) \leftarrow a(i) \times a(i)$ is implemented both in serial and parallel form. Note that a function encapsulates the operation (this is not the only approach).
  - Note that we use `using namespace tbb`. This avoids having to include the prefix `tbb` before each identifier used by the `tbb` library (e.g.: *parallel_for, blocked_range*).

- Compile this application:
  `g++ basic.cpp -ltbb -o basic` ↵

- Execute this application:
  `./basic` ↵

SECOND ACTIVITY: CEILING OPERATOR IN A VECTOR

- Given an n-element integer vector, we compute $b(i) \leftarrow \left\lceil \frac{a(i)}{2} \right\rceil$ for every element, where $a(i)$ and $b(i)$ are integer values.
- Example: `a = [0 3 7 12 18 25 33 42 52 63]`
  Result: `b = [0 2 4  6  9 13 17 21 26 32]`

- This example will introduce the concept of lambda expressions for the *parallel_for* template function. Lambda expressions allow us to implement parallel operations without having to define a class or a structure (unlike in the First Activity).
- In order to use lambda expressions, we need the Version 11.0 of the Intel® C++ Compiler. This requires the use of the modifier `–std=c+11` when compiling.
- To show a different approach than in the first example, functions are not used here.
- Note that we use `using namespace std`. This avoids having to include the prefix `std` before each identifier used by the `std` library. As a result, we need to use the prefix `tbb` for *parallel_for*.
- In this example, we will provide two different approaches:
  - ✓ Normal lambda expressions: The iteration space can be customized. Two variations will be shown: local variables captured by value and captured by reference.
  - ✓ Compact lambda expressions: Very useful as they are short-hand declarations for *parallel_for*. However, the iteration space only supports looping is over a consecutive range of integers.

Normal lambda expressions

- We will implement the ceiling operator applied to every element in parallel.
  - ✓ Version 1:

```
tbb::parallel_for( tbb::blocked_range<size_t>(0,n), [=](const tbb::blocked_range<size_t>& r) {
          for(size_t k=r.begin(); k!=r.end(); ++k) // 0 <= k < n
          if (a[k] == 0) b[k] = 0;
                 else b[k] = (a[k]-1)/DV + 1;  // b[k] = ceil(a[k]/DV)
              }
    );
```

- ▫ The *parallel_for* template indicates the iteration space (`0` to `n-1`), the type of the range index, and the operation itself.
- ▫ Note that we need to define the loop operation inside.
- ▫ The expression `[=]` creates a function object. When the local variables (`a,b,n`) are declared outside the lambda expression, they are "captured" as fields inside the function object. It specifies that capture is by value. It also means that the range index `r` needs the modifier & (we directly specify that it is pointer).
  - · This means that local variables cannot be modified (passed by value), otherwise the compiler will issue an error. Note however that we can modify arrays as it is interpreted that we are passing the pointer values.

- ✓ Version 2 (preferred):

```
tbb::parallel_for ( tbb::blocked_range<size_t>(0,n), [&](const tbb::blocked_range<size_t> r) {
         for(int k=r.begin(); k!=r.end(); ++k) // 0 <= k < n
            if (a[k] == 0) b[k] = 0;
            else b[k] = b[k] = (a[k]-1)/DV + 1; // b[k] = ceil(a[k]/DV) // DV = 2
     }
   );
```

- ▫ The *parallel_for* template indicates the iteration space (`0` to `n-1`), the type of the range index, and the operation itself.
- ▫ The operator `[&]` indicates that the local variables (`a,b,n`) are captured by reference. It also means that the range index `r` does not need any modifier.

Compact lambda expressions

- The compact form support only unidimensional iteration spaces of consecutive integers.
- There are also two versions: `[=]` and `[&]`. As already mentioned, they only specify whether local variables declared outside the lambda expression are captured as fields inside the function objects either by value (`[=]`) or by reference (`[&]`). Note that the index (`k`) is just a variable and not a range like in the normal lambda expressions.

- We will implement the ceiling operator applied to every element in parallel.
  - ✓ Version 1:

```
tbb::parallel_for(size_t(0), size_t(n), [=] (size_t k) { // 1 <= k < n
          if (a[k] == 0) b[k] = 0;
          else b[k] = b[k] = (a[k]-1)/DV + 1; // b[k] = ceil(a[k]/DV)
    } );
```

  - ✓ Version 2:

```
tbb::parallel_for(size_t(0), size_t(n), [&] (size_t k) { // 1 <= k < n
          if (a[k] == 0) b[k] = 0;
          else b[k] = b[k] = (a[k]-1)/DV + 1; // b[k] = ceil(a[k]/DV)
    } );
```

- Application file: `myceil.cpp`
  - ✓ In this implementation, the operation $b(i) \leftarrow \left\lceil \frac{a(i)}{2} \right\rceil$ is implemented in parallel form for both the lambda expressions (normal, compact).

- Compile this application:
  `g++ -std=c++11 myceil.cpp -ltbb -o myceil` ↵

- Execute this application: `./myceil` ↵

THIRD ACTIVITY: ELEMENT-WISE CONSTANT MULTIPLIER

- Given an n-element integer vector, we want to compute $a(i) \leftarrow a(i) \times 2$ for every element in the vector, where each element $a(i)$ is a real value.
- Example: `a = [1.2 2.3 3.6  6.7 0.1 0.2 2.1 0.7  5.1 0.3]`
  Result: `a = [2.4 4.6 7.2 13.4 0.2 0.4 4.2 1.4 10.2 0.6]`

- This straightforward example illustrates four ways to use the lambda expressions for *parallel_for*:
  - ✓ Normal lambda expression:
    - Inside the main function.
    - Inside a function:

```
void ParallelApplyFun ( float a[], size_t n ) {
    tbb::parallel_for( tbb::blocked_range<int>(0,n), [&](tbb::blocked_range<int> r)
    {
        for (int i=r.begin(); i<r.end(); ++i)
        {
            a[i] = a[i]*2;
        }
     });
}
```

  - ✓ Compact lambda expression:
    - Inside the main function
    - Inside a function:

```
void ParallelApplyFun_c ( float a[], size_t n ) {
    tbb::parallel_for(size_t(0), n, [&] (size_t i) {
         a[i] = a[i]*2;
    } );
}
```

- In order to use lambda expressions, we need the Version 11.0 of the Intel® C++ Compiler. This requires the use of the modifier `-std=c+11` at compilation.
- Note that we use `using namespace std;`. This avoids having to include the prefix `std` before each identifier used by the `std` library. As a result, we need to use the prefix `tbb` for *parallel_for*.

- Application files: `simple.cpp`
  - ✓ In this implementation, the operation $a(i) \leftarrow a(i) \times 2$ is implemented in parallel form for both the lambda expressions (normal, compact) inside a user-defined function and inside the main function.

- Compile this application:
  `g++ -std=c++11 simple.cpp -ltbb -o simple` ↵

- Execute this application:
  `./simple` ↵

FOURTH ACTIVITY: 3-ELEMENT MOVING AVERAGE

- Given an n-element integer vector $\vec{a}$, we want to compute the 3-element moving average. If $a(i)$ is an element of the vector, the 3-element moving average is given by:

$$f(i) \leftarrow \frac{a(i-1) + a(i) + a(i+1)}{3}$$

  - ✓ The moving average is usually a central moving average that can be computed using data equally spaced on either side of a central value (this needs the number of elements in the window to be odd).
  - ✓ In the formula, $i = 0, \dots n-1$. When the elements are not available (at the borders), we only use the available elements:

$$f(0) \leftarrow \frac{a(i) + a(i+1)}{2}, \qquad f(n-1) \leftarrow \frac{a(i-1) + a(i)}{2}$$

- TBB: We use the compact lambda expression for *parallel_for* here:

```
tbb::parallel_for(int(0), int(n), [&] (int k) { // 0 <= k <= n-1
        if (k == 0) g[k] = (a[k] + a[k+1]) /2;
        else if (k == n-1) g[k] = (a[k-1] + a[k]) /2;
        else g[k] = (a[k-1] + a[k] + a[k+1]) /3;
    } );
```

- Fig. 2 depicts an example. The original data (102 data points) is plotted as a series of dots. The moving average (3 elements) smooths short-term fluctuations and highlight longer-term trends.

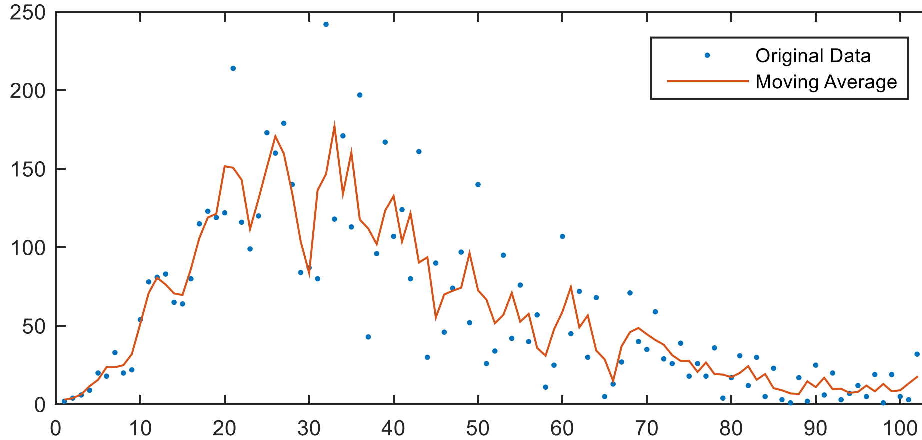

Figure 2. Three-element moving average for a 102-point dataset.

- In order to use lambda expressions, we need the Version 11.0 of the Intel® C++ Compiler. This requires the use of the modifier `-std=c+11` at compilation.
- Note that we use `using namespace std;`. This avoids having to include the prefix `std` before each identifier used by the `std` library. As a result, we need to use the prefix `tbb` for *parallel_for*.

- Application files: `mov_avg.cpp`
  - ✓ Here, the 3-element moving average is implemented both in serial and parallel (compact lambda expression) forms.

- Compile this application: `g++ -std=c++11 mov_avg.cpp -ltbb -o mov_avg` ↵
- Execute this application: `./mov_avg <# of elements in vector>` ↵
  - ✓ Example: `./mov_avg 10` ↵

- Table I lists the computation times for different sizes. There is improvement in processing time for datasets of large size n. A main reason for this is the relative simplicity of the computation applied to each element.

TABLE I. COMPUTATION TIMES (US) FOR DIFFERENT DATASET SIZES (n) FOR THE 3-ELEMENT MOVING AVERAGE. DELL INSPIRON

n	Processing Time (us)		n	Processing Time (us)	
	Sequential	TBB		Sequential	TBB
10	1	499	100,000	3124	1991
50	3	734	500,000	9965	6851
100	3	779	1,000,000	17262	13822
1,000	19	818	2,000,000	34520	27665
10,000	235	1014	5,000,000	84136	66454
50,000	1557	1380	10,000,000	165654	124882

FIFTH ACTIVITY: VECTOR OPERATIONS

- We perform element-wise add and multiply for two n-element vectors with elements $a(i)$ and $b(i)$.

$$c(i) \leftarrow a(i) + b(i)$$
$$d(i) \leftarrow a(i) \times b(i)$$

- We use the compact lambda expression here. 3 approaches:
  - ✓ Element-wise operations defined inside the *parallel_for* inside the main function.

```
tbb::parallel_for(int(0), int(n), [&] (int k) { // 0 <= k <= n-1. [0,n): half-open range
        c[k] = a[k] * b[k];
        d[k] = a[k] + b[k];
    } );
```

  - ✓ Element-wise operations inside the *parallel_for* are defined in a function.

```
void point_op (float x, float y, float *z, float *w)
{
   z[0] = x*y;
   w[0] = x+y;
}

tbb::parallel_for(int(0), int(n), [&] (int k) { // 0 <= k <= n-1
         point_op(a[k], b[k], &c[k], &d[k]);
   } );
```

  - ✓ All element-wise operations are defined in a function.

```
Parallel_point_op(a,b,c,d,n);

void Parallel_point_op (float *a, float *b, float *c, float *d, int n) {
    tbb::parallel_for(int(0), int(n), [&] (int k) { // 0 <= k <= n-1
           point_op(a[k], b[k], &c[k], &d[k]);        // Approach 1
           // c[k] = a[k] * b[k]; d[k] = a[k] + b[k]; // Approach 2
    } );
}
```

- In order to use lambda expressions, we need the Version 11.0 of the Intel® C++ Compiler. This requires the use of the modifier `-std=c+11` at compilation.
- Note that we use `using namespace std;`. This avoids having to include the prefix `std` before each identifier used by the `std` library. As a result, we need to use the prefix `tbb` for *parallel_for*.

- Application files: `vector_op.cpp`
  - ✓ In this implementation, the 3-element moving average is implemented both in serial and parallel (compact lambda expression) forms.

- Compile this application:

```
g++ -std=c++11 vector_op.cpp -ltbb -o vector_op ↵
```

- Execute this application:

```
./vector_op ↵
```

SIXTH ACTIVITY: GRAYSCALE MORPHOLOGICAL OPERATIONS (DILATION, EROSION)

GRAYSCALE IMAGE MORPHOLOGY

- Set of image processing operations that applies a structuring element to an input grayscale image, generating an output grayscale image of the same size. The value of an output pixel is based on a comparison of the corresponding pixel in the input image with its neighbors. The neighborhood is also called the structuring element. By choosing the size and shape of the structuring element, we can construct a morphological operation that is sensitive to specific shapes in the input image. The morphological operation is applied to each input pixel independently. Among the applications, we can list contrast enhancement, texture description, edge detection, and thresholding.
- Common morphological operations: dilation, erosion, opening, closing, smoothing, edge detection, top-hat.

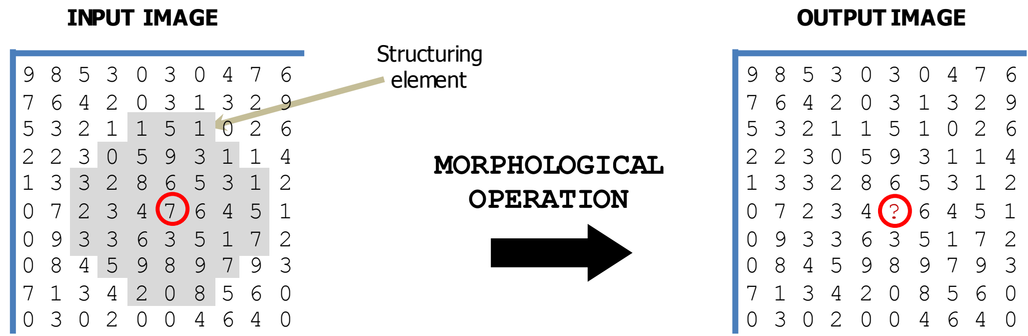

Figure 3. Structured Element (SE) applied to an input pixel (the circled one). The SE is centered around this input pixel. The operation applied depends on the specific morphological operation.

- **Structuring element (SE)**: It defines the neighborhood over which we apply a morphological operation to the input pixel of interest (which is located at the origin of the SE). We consider cases where the origin of the SE is easily identifiable. The structuring element can have any shape and size. Fig. 4 depicts several SEs of different shapes, where the center of the structuring element (circled in red) is the input pixel over which we want to apply a morphological operation.
  - ✓ A SE is like a 'mask' defined by a matrix, whose values are '0' or '1'. A value of '1' indicates that the pixel in that location belongs to the neighborhood and as such it is considered in the computation of the morphological operation. A '0' indicates that the pixel in that local is not part of the neighborhood.

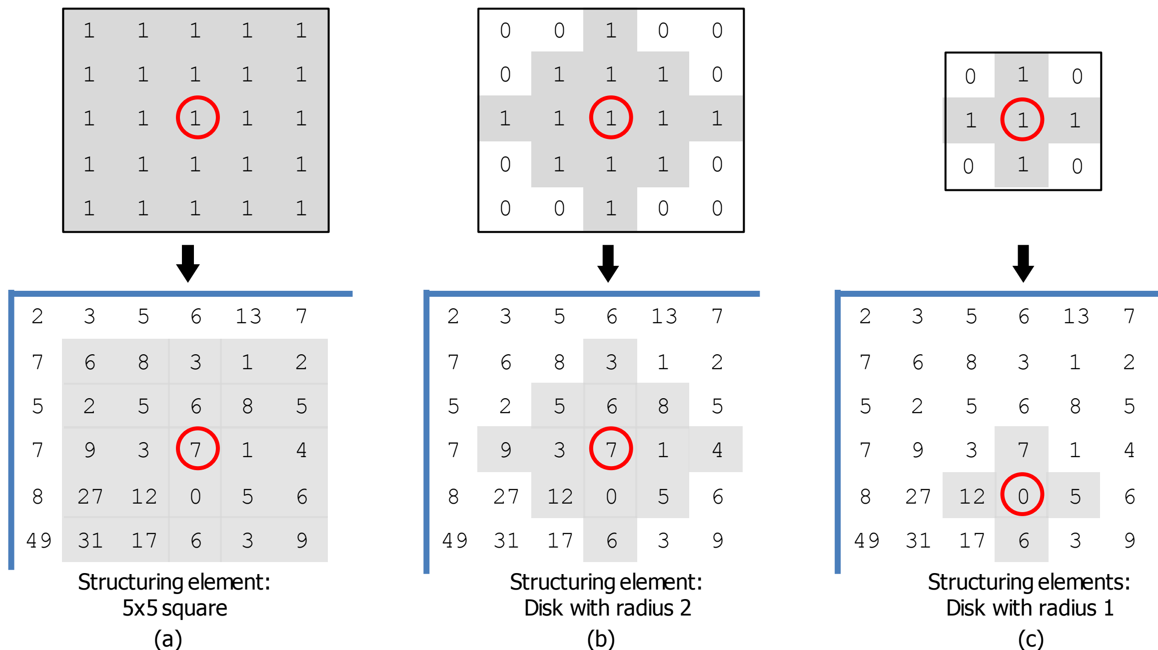

Figure 4. Three different structured elements (SEs). They define the neighborhood of the input pixel over which we apply a specific operation.

- **Dilation**: The grayscale dilation of an image involves assigning to each pixel, the maximum value found over the neighborhood of the structuring element. The dilated value of a pixel `I(i,j)` is the maximum value of the image in the neighborhood defined by the SE when its origin is at `I(i,j)`.
- **Erosion**: The grayscale erosion of an image involves assigning to each pixel, the minimum value found over the neighborhood of the structuring element. The eroded value of a pixel `I(i,j)` is the minimum value of the image in the neighborhood defined by the SE when its origin is at `I(i,j)`.

* *This material is based upon work supported by Intel® Corporation*

- Fig. 5 depicts the operation of dilation and erosion applied to a grayscale image with the given Structural Element. Note how dilation brighten pixels to the boundaries of bright areas, while erosion darkens pixels from the boundary of bright areas.

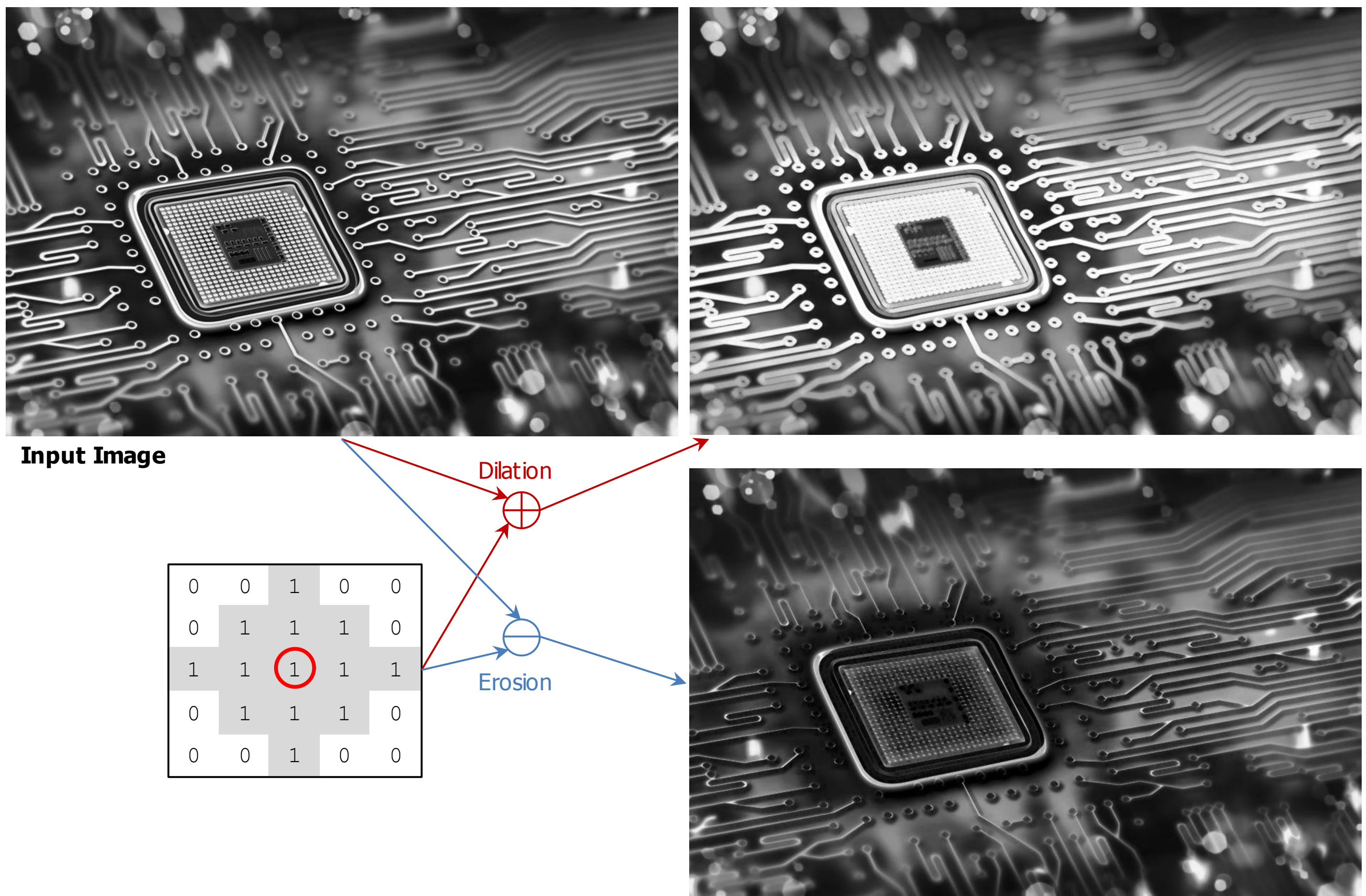

Figure 5. Dilation and erosion applied to a grayscale input image. The Structural Element (SE) is indicated in the figure.

- TBB: We use the compact lambda expression for *parallel_for* here. The dilation/erosion operation is implemented as a function `im_morph_tbb (MORPHO_args morpho_data, size_t t)`:

```
int im_morph_tbb (MORPHO_args morpho_data, size_t t) { // t = 1: Dilation. t = 2: Erosion
   int sX, sY, kX, kY;
   int *I, *O, *K; // these are provided as linear arrays
   // morpho_data.I: input image matrix represented as a 1D vector (raster-scan)
   I = morpho_data.I; O = morpho_data.O; sX = morpho_data.SX; sY = morpho_data.SY;
   K = morpho_data.K; kX = morpho_data.KX; kY = morpho_data.KY;

   if (!I || !O || !K) return 0;
   if (sX <= 0 || sY <= 0 || kX <= 0 || kY <= 0) return 0;

   if (t == 1) { // DILATION
     tbb::parallel_for (int(0), int(sY), [&] (int i) { // 0 <= i <= SY-1
          tbb::parallel_for (int(0), int(sX), [&] (int j) { // 0 <= j <= SX-1
                 O[sX*i + j] = maxI(morpho_data,i,j);      } ); // Fun(i,j) = maxI(i,j)
        } );
     }
     else { // t==2: EROSION
    tbb::parallel_for (int(0), int(sY), [&] (int i) { // 0 <= i <= SY-1
          tbb::parallel_for (int(0), int(sX), [&] (int j) { // 0 <= j <= SX-1
                 O[sX*i + j] = minI(morpho_data,i,j);      } ); // Fun(i,j) = minI(i,j)
        } );
     }
  return 1;
}
```

✓ Note that we are using nested *parallel_for*. We could have used one *parallel_for* (since the input image is specified as a linear array), but we prefer to illustrate the usage of nested *parallel_for*.

- The function `Fun` applied to every element of the array is `Fun(I(i,j))`. The operation at every index is independent from each other. To avoid race conditions, the function `Fun(I(i,j))` is implemented as a function in C++ for both dilation (`Fun(I(i,j)) = maxI(I(I,j))`) and erosion (`Fun(I(i,j)) = minI(I(I,j))`).

```
int minI (MORPHO_args morpho_data, int i, int j) {
  int min, m, n, kCX, kCY, kX, kY, sX, sY, i_I, j_I;
  int *I, *K;

  I = morpho_data.I; sX = morpho_data.SX; sY = morpho_data.SY;
  K = morpho_data.K; kX = morpho_data.KX; kY = morpho_data.KY;
  kCX = kX / 2; kCY = kY / 2; // find center position of kernel (half of kernel size, flooring)

  min = I[sX*i + j];
  for (m = 0; m < kY; m++) // SE: rows
     for (n = 0; n < kX; n++) { // SE: columns
        if (K[kX*m + n] == 1) { // out-of-bounds input samples are not considered
            i_I = i + m-kCY;  j_I = j + n-kCX;
            if (i_I >= 0 && i_I < sY && j_I >= 0 && j_I < sX)
                 if ( I[sX*i_I + j_I] <= min )  min = I [sX*i_I + j_I];
        }
     }
  return min;
}

int maxI (MORPHO_args morpho_data, int i, int j) {
  int max, m, n, kCX, kCY, kX, kY, sX, sY, i_I, j_I;
  int *I, *K;

  I = morpho_data.I; sX = morpho_data.SX; sY = morpho_data.SY;
  K = morpho_data.K; kX = morpho_data.KX; kY = morpho_data.KY;
  kCX = kX / 2; kCY = kY / 2;  // find center position of kernel (half of kernel size, flooring)

  max = 0;
  for (m = 0; m < kY; m++) // SE: rows
     for (n = 0; n < kX; n++) { // SE: columns
        if (K[kX*m + n] == 1) {  // out-of-bounds input samples are not considered
           i_I = i + m-kCY; j_I = j + n-kCX;
           if (i_I >= 0 && i_I < sY && j_I >= 0 && j_I < sX)
              if ( I[sX*i_I + j_I] >= max )  max = I [sX*i_I + j_I];
        }
     }
  return max;
}
```

- Application files: `morpho.cpp, morpho_fun.cpp, morpho_fun.h, Makefile`
  - ✓ Code Structure:
    - Determination of whether to apply dilation or erosion. Initialization of structural element.
    - Initialization of input image I by reading from binary file (`uchip.bif`).
    - Sequential computation (using a function) of grayscale dilation/erosion.
    - TBB Computation (using compact lambda expression for *parallel_for*) of grayscale dilation/erosion
    - Store result (output matrix) on a binary file (`uchip_d.bof` for dilation) and (`uchip_e.bof` for erosion).
  - ✓ We measure the processing time (us) of both the serial and parallel implementation using `gettimeofday()`.

- Compile this application: `make all ↵`
- Execute this application: `./morpho <modifier> ↵`
  - ✓ Two execution possibilities:
    - `./morpho 1 ↵` (dilation): Output binary file: `uchip_d.bof`
    - `./morpho 2 ↵` (erosion): Output binary file: `uchip_e.bof`
  - ✓ You can use MATLAB® to verify that your results (output images) are correct.
    - Files: `morpho.m, uchip.jpg`

- Table I lists the computation times. There is improvement in processing time when using TBB.

TABLE I. COMPUTATION TIME (US) OF DILATION/EROSION. DELL INSPIRON

	Computation Time (us)	
	Sequential	TBB
Dilation	46003	31733
Erosion	46248	29275

EXERCISES

- Given an n-element vector, compute the 7-element moving average using *parallel_for* to run the computations in parallel.
- Perform grayscale morphological opening (erosion + dilation) and closing (dilation + erosion) on an image.