Using Intel® Math Kernel Library and Intel® Integrated Performance Primitives in the Microsoft® .NET® Framework
INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request. Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families.


* Other names and brands may be claimed as the property of others.

Copyright© 2009, Intel Corporation. All rights reserved.
# Table of Contents

Introduction ................................................................................................................. 4

Microsoft .NET Framework Overview ........................................................................ 4
  .NET Framework Terminology .................................................................................. 4
  .NET Framework Interoperability Mechanisms ....................................................... 5

Intel® MKL Overview .................................................................................................... 8
  Components – Implementation Details ................................................................. 9
  BLAS and LAPACK ................................................................................................. 9
  VML ....................................................................................................................... 9
  FFT ..................................................................................................................... 10

Intel® IPP Overview .................................................................................................... 12
  C# interface to Intel® IPP. Namespace and Structures ........................................... 13
  Intel® IPP Components - Image Processing Sample .............................................. 14

Intel® MKL and Intel® IPP Performance from C# .................................................... 18

Conclusion .................................................................................................................. 18

Appendix A. .................................................................................................................. 19
  Deferred Mode Image Processing and C++ Interoperability .................................. 19
    C++ Interoperability ............................................................................................. 19
    Mixed Native and Managed Assembly ................................................................ 19
    P/Invoke and C++ Interoperability ..................................................................... 19
    .NET Wrappers for Native C++ DMIP/UIC High Level API ............................ 20
    Performance of DMIP Library ............................................................................. 22

References ................................................................................................................... 23
Introduction

This paper is intended to educate Intel® Math Kernel Library (Intel® MKL) and Intel® Integrated Performance Primitives (Intel® IPP) users on the basics of calling these libraries from .NET Framework languages such as C#. The most important consideration is how to manage the calls between the managed .NET application and the unmanaged Intel® MKL and Intel® IPP Libraries. Figure 1 provides a high level perspective.

Intel® Performance Libraries such as Intel® MKL and Intel® IPP are unmanaged code libraries. They are written in native programming languages and compiled to machine code which can run on a target computer directly.

In this paper we consider the Platform Invocation services (P/Invoke) .NET interoperability mechanism specifically for calling the Intel® Performance Libraries from C#. We describe the basic concepts such as passing callback functions and arguments of the different data types from managed to unmanaged code and the pinning of array arguments.

A brief introduction into Intel® MKL and Intel® IPP and their components is given to illustrate .NET interoperability features. Detailed examples can be found in the Appendix A.

The reference section contains links to the Microsoft* online documentation (MSDN) for those interested in .NET features in more detail.

Microsoft .NET Framework Overview

.NET Framework Terminology

Microsoft .NET framework is a managed runtime environment for developing the applications that target the common language runtime (CLR) layer. This layer consists of the runtime execution services needed to develop various types of software applications, such as ASP .NET, Windows forms, XML Web services, distributed applications and others. Compilers targeting the CLR must conform to
the common language specification (CLS) and common type system (CTS), which are sets of language features and types common among all the languages. These specifications enable type safety and cross-language interoperability. It means that an object written in one programming language can be invoked from another object written in another language targeting the runtime.

C# is a programming language designed as the main language of the Microsoft .NET Framework. It compiles to a Common Intermediate language (CIL) like all the other languages compliant to the .NET Framework. CIL provides a code that runs under control of the CLR. This is managed code. All codes that run outside the CLR are referred to as unmanaged codes. CIL is an element of an assembly, a collection of types and resources that are built to work together and form a logical unit of functionality. Assemblies are the building blocks of the .NET Framework applications. They are stored in the portable executable (PE) files and can be a DLL or EXE [6]. Assemblies also contain metadata, the information used by the CLR to guarantee security, type safety, and memory safety for code execution.

The .NET Framework's garbage collector (GC) service manages the allocation and release of memory for the managed objects in the application. The garbage collector checks for objects in the managed heap that are no longer used by the application and performs the operations necessary to reclaim their memory. The data under control of the garbage collector is managed data.

C# code that uses pointers is called unsafe code. The keyword unsafe is a required modifier for the callable members such as properties, methods, constructors, classes, or any block of code. Unsafe code is a C# feature for performing memory manipulation using pointers. Use the keyword fixed (pin the object) to avoid movement of the managed object by the GC.

---

**Note:** Unsafe code must be compiled with the /unsafe compiler option.

### .NET Framework Interoperability Mechanisms

The CLR supports the Platform Invocation Service (P/Invoke) that allows mapping a declaration of a managed method to unmanaged methods. The resulting declaration describes the stack frame but assumes the external method body from a native DLL [5].

#### Platform Invocation (P/Invoke) Service

The Platform Invocation Service called P/Invoke enables managed code to call C-style unmanaged functions in native DLLs. P/Invoke can be used in any .NET Framework-compliant language. It is important to be familiar with the attributes DllImport, MarshalAs, StructLayout and their enumerations to use P/Invoke effectively.

When a P/Invoke call is initiated to call an unmanaged function in the native DLL, the P/Invoke service performs the following steps:

1. Locates the DLL specified by the `DllImport` attribute by searching either in the working directory or in the directories and sub-directories specified in the PATH variable, and then loads the DLL into the memory
2. Finds the function declared as static extern in the DLL loaded to memory
3. Pushes the arguments on the stack by performing marshalling and if required, using the attributes MarshalAs and StructLayout
4. Disables pre-emptive garbage collection
5. Transfers the control to the unmanaged function.

**Declare static extern Method with theDllImport Attribute**

An unmanaged method must be declared as `static extern` with the `DllImport` attribute. This attribute defines the name of a native shared library (native DLL) where the unmanaged function is located. The attribute `DllImport` and function specifiers `static extern` specify the method that is used by the .NET Framework to create a function and to marshal data. The `DllImport` attribute has parameters to specify correspondence rules between managed and native methods, such as `CharSet` (Unicode or Ansi), `ExactSpelling` (true or false), `CallingConvention` (cdecl or StdCall), `EntryPoint` [5], [15].

In the simplest case the managed code can directly call a method `foo()` declared as:

```csharp
DllImport("custom.dll")
static extern double
foo(double a, double b);
```

**Marshalling for the Parameters and Return Values**

The .NET Framework provides an interoperability `marshaller` to convert data between managed and unmanaged environments. When managed code calls a native method, parameters are passed on the call stack. These parameters represent data in both the CLR and native code. They have the managed type and the native type.

Some data types have identical data representations in both managed and unmanaged code. They are called isomorphic, or blittable data types [4]. They do not need special handling or conversion when passed between managed and unmanaged code. Basic data types of this kind are: float/double, integer, and one-dimensional arrays of isomorphic types. These are common types in both Intel® MKL and Intel® IPP.

However, some types have different representations in managed and unmanaged code. These types are classified as non-isomorphic, or non-blittable data types and require conversion, or marshalling. The following table presents some non-isomorphic types commonly used in the .NET Framework.

<table>
<thead>
<tr>
<th>Managed</th>
<th>Unmanaged</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>BOOL, Win32 BOOL, or Variant</td>
</tr>
<tr>
<td>Char</td>
<td>CHAR, or Win32 WCHAR</td>
</tr>
<tr>
<td>Object</td>
<td>Variant, or Interface</td>
</tr>
<tr>
<td>String</td>
<td>LPStr, LPWStr, or BStr</td>
</tr>
<tr>
<td>Array</td>
<td>SafeArray</td>
</tr>
</tbody>
</table>

*Table 1. Mapping of several data types, unmanaged-managed environment*
In some cases the default marshalling can be used. But not all parameters or return values can be marshalled with the default mechanism. In such cases the default marshalling can be overridden with the appropriate marshalling information. Marshalling includes not only conversion of the data type, but other options such as the description of the data layout and direction of the parameter passing. There are some attributes for these purposes: MarshalAs, StructLayout, FieldOffset, InAttribute, and OutAttribute. (See more details in [5], [7], [8], [9], [10].)

MarshalAsAttribute and Marshal class in the System.Runtime.InteropServices namespace can be used for marshalling non-isomorphic data between managed and unmanaged code.

**Pinning Arrays and Other Objects**

All objects of the .NET Framework are managed by the garbage collector (GC). The GC can relocate an object in memory asynchronously. If managed code passes to native code a reference to a managed object, this object must be prevented from being moved in the memory. Its current location must be locked (or pinned) for the life of the native call.

Pinning can be performed manually or automatically.

**Manual, or explicit, pinning can be performed by creating GCHandle:**

```csharp
GCHandle pinnedObj = GCHandle.Alloc(anObj, GCHandleType.Pinned);
```

Pinning can also be performed using the `fixed` statement in unsafe code. An `/unsafe` compilation option is required in this case [12].

The other method is automatic pinning. When the runtime marshaller meets managed code that passes an object to the native code, it automatically locks the referenced objects in memory for the duration of the call [13].

Declarations are similar for both methods:

```csharp
// fixed statement in unsafe code
[DllImport("custom.dll")]
unsafe static extern float
foo(float *x);

// automatic pinning
[DllImport("custom.dll")]
static extern float
foo(float[] x);
```

The automatic pinning enables calling native methods in a usual manner and does not require the `/unsafe` option during compiling.

Note that aggressive pinning of short-lived objects is not good practice because the GC cannot move a pinned object. This can cause the heap to become fragmented, which reduces the available memory [12], [14].

**Callback Function**

A callback function is a managed or unmanaged function that is passed to an unmanaged DLL function as a parameter. To register a managed callback that an unmanaged function uses, declare a
delegate with the same argument list and pass an instance of it through P/Invoke. On the unmanaged side, it appears as a function pointer (see [18] and [21]).

The CLR automatically pins the delegate for the duration of the native call. Moreover, there is no need to create a pinning handle for the delegate for the asynchronous call: in this case the unmanaged function pointer actually refers to a native code stub dynamically generated to perform the transition and marshaling. This stub exists in fixed memory outside of the GC heap. The lifetime of the native code stub is directly related to the lifetime of delegate [22], [23].

The delegate instance, which is passed to unmanaged code, employs the StdCall calling convention. An attribute to specify Cdecl calling convention is available only in the .Net Framework v2.0+. (See VML for more details.)

**Intel® MKL Overview**

Intel® MKL includes a wide variety of highly optimized math functions which operate on vectors and matrices. Both Fortran and C interface to Intel® MKL functions and routines are available.

In this paper we consider the following groups of Intel® MKL routines:

- Basic Linear Algebra Subroutines (BLAS)
- Linear Algebra Package (LAPACK)
- Vector Mathematical Functional Library (VML)
- Fast Fourier Transform functions (FFT)
- Parallel Direct Sparse Solver Interface (PARDISO)

Most of the domains support four data types: float, double, complex, and double complex. A complex number is stored as a two-element array: the real part of the number is the first element of the array, and the imaginary part is the second element. Vectors and matrices are represented by 1- and 2-dimensional column-major arrays. Some components allow the use of row-major arrays also. For more details see the "Intel® MKL Reference Manual" [1].

Users pass arrays to Intel® MKL routines as pointers to memory locations. Intel® MKL routines may allocate and de-allocate memory for internal purposes. Because sequential and parallel versions of Intel® MKL routines are available, all Intel® MKL functions work correctly when simultaneously executed by multiple threads. Some components may require creating a job descriptor first and passing it as an argument in further function calls. Some Intel® MKL components also use a callback function to handle errors.

Intel® MKL provides libraries to support both static and dynamic linking. To call Intel® MKL from a .NET language, a custom dynamic library must be built. For details see "Intel® MKL User’s Guide" [2], chapters 3 and 5. The name custom.dll is used for the Dllimport library in the Intel® MKL examples below.
Note: Default calling convention in Intel® MKL is Cdecl, default calling convention in the .NET Framework is StdCall.

Components - Implementation Details

BLAS and LAPACK

Intel® MKL implements CBLAS - the C interface to the BLAS. CBLAS allows using both C- and Fortran-style matrices (row- and column-major). As we mentioned above (Pinning Arrays and Other Objects), two ways of passing arrays are equally possible. We suggest using automatic pinning as it allows calling native methods in a usual C# manner and does not require the /unsafe option during compiling.

Every Interop call introduces some overhead. The amount of overhead depends on the types of parameters. One more important performance related aspect of calling unmanaged code is a full stack walk to check permissions. The attribute SuppressUnmanagedCodeSecurity allows checking permissions only for a caller and disables the full stack walk (see [8] and [24]). The benefit of using the SuppressUnmanagedCodeSecurity attribute is more essential for “fast” calls, for example, for calls of Intel® MKL functions with small dimensions.

```csharp
[SUPPRESSUNMANAGEDCODESECURITY]
[DllImport("custom.dll", CallingConvention=CallingConvention.Cdecl)]
static extern void cblas_dgemm(int Order, int TransA, int TransB,
                                 int M, int N, int K, double alpha, [In] double[,] A, int lda, [In]
                                 double[,] B, int ldb, double beta, [In, Out] double[,] C, int ldc);
```

LAPACK has only Fortran interfaces. That means scalars are passed as references, and only Fortran-style (column-major) matrices are allowed. Many LAPACK routines use work arrays. Some array arguments may be not referenced for certain conditions.

```csharp
[SUPPRESSUNMANAGEDCODESECURITY]
[DllImport("custom.dll", CallingConvention=CallingConvention.Cdecl)]
static extern void dgeev(ref char jobvl, ref char jobvr,
                         ref int n, [In, Out] double[,] a, ref int lda,
                         [Out] double[,] wr, [Out] double[,] wi,
                         [Out] double[,] vl, ref int ldvl, [Out] double[,] vr, ref int ldvr,
                         [In, Out] double[,] work, ref int lwork, ref int info);
```

In this LAPACK routine vr is not referenced if jobvr = 'N', and vl is not referenced if jobvl = 'N'. The user can pass a “null” value in the case unreferenced arguments. It does not matter for automatic pinning but it may require additional efforts to check null values in case of passing arrays as pointers with fixed keyword within unsafe block.

VML

The Vector Math Library (VML) computes mathematical functions on vector arguments. VML sets a thread-private global variable after each function call. Special service functions get, set, and clear the
error status. VML gives a possibility to fix an error and resume computation. To do this, a callback function must be set. It is called from a VML function on each error. The input structure for the callback function contains all necessary information about the error encountered. The passed result values may be corrected and passed back to resume computation.

Although the VML callback function is invoked after the set function keeps pointer to the callback function within unmanaged memory and returns, it is not required to pin delegate manually. As described above the unmanaged function pointer actually refers to a native code stub outside of the GC heap. The lifetime of the native code stub is related to the lifetime of managed delegate.

The code example below shows callback declarations and marshalling the VML error structure.

Note that the UnmanagedType attribute is a feature of the .NET Framework v2.0+

```csharp
using System;
using System.Runtime.InteropServices;

public struct VmlErrorContext
{
    public int iCode;   /* Error status value */
    public int iIndex;  /* Bad array element, dimension or pointer */
    public double dbA1; /* Error argument 1 */
    public double dbA2; /* Error argument 2 */
    public double dbR1; /* Error result 1 */
    public double dbR2; /* Error result 2 */
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst=64)]
    public string cFuncName; /* Function name */
    int iFuncNameLen; /* Length of function name*/
}

[DllImport("custom.dll", CallingConvention=CallingConvention.Cdecl)]
internal static extern int DftiCreateDescriptor(ref IntPtr desc, int precision, int domain, int dimension, int length);

FFT

The Intel® MKL FFT functions are accessible through the Discrete Fourier Transform Interface (DFTI). This interface is rather simple and consists of a few functions. Computing FFT with DFTI requires the following sequence of steps:

- Descriptor must be associated with a descriptor handle by a call to the function DftiCreateDescriptor. Using IntPtr type for descriptor instead of the pointers simplifies the interface.
The descriptor remains an opaque structure, the handle to the descriptor is passed to all DFTI functions.

- Descriptor may optionally be adjusted with DftiGetValue and DftiSetValue functions. Versatility of DFTI relies on a variable number of arguments of the functions. C# provides no documented way to pass a variable number of arguments via P/Invoke, but there is a working solution to use C# undocumented __arglist keyword in the declaration.

```csharp
[DllImport("custom.dll", CallingConvention=CallingConvention.Cdecl)]
internal static extern int DftiSetValue(IntPtr desc, int config_param, int __arglist);
```

and in the function call,

```csharp
DftiSetValue(desc, parameter, __arglist(value));
```

where `value` could have either `int` or `double` type.

- Descriptor must be committed with the function DftiCommitDescriptor

```csharp
[DllImport("custom.dll", CallingConvention=CallingConvention.Cdecl)]
internal static extern int DftiCommitDescriptor(IntPtr desc);
```

- Now it can be used in the computational functions DftiComputeForward and DftiComputeBackward

```csharp
[DllImport("custom.dll", CallingConvention=CallingConvention.Cdecl)]
internal static extern int DftiComputeForward(IntPtr desc, [In] double[] x_in, [Out] double[] x_out);

[DllImport("custom.dll", CallingConvention=CallingConvention.Cdecl)]
internal static extern int DftiComputeBackward(IntPtr desc, [In] double[] x_in, [Out] double[] x_out);
```

- When not needed, the descriptor must be discarded by a call to the function DftiFreeDescriptor.
Intel® MKL PARDISO* is a well known parallel direct sparse solver. It uses advanced parallel algorithms to solve general sparse systems of equations on multi-core systems and has three main phases of execution: matrix reordering, LU (lower/upper) factorization of the input matrix, and the solution of the factored system. It has possibility to make factorization and solving in different steps. This way it is somehow similar to DFTI. But unlike DFTI all work can be done via one call. PARDISO supports 9 types of real and complex matrices which could be solved using single or double precisions. This component contains only one method with a FORTRAN interface where scalars are passed as references. The corresponding C# declaration is simple:

```csharp
[DllImport("custom.dll", CallingConvention=CallingConvention.Cdecl)]
internal static extern int DftiFreeDescriptor(ref IntPtr desc);
```

Intel® IPP Overview

Intel® IPP is a low level software library. It provides a set of basic functions highly optimized for the IA-32, IA-64, and Intel® 64 architectures. The use of the library significantly speeds up a wide variety of software in different application areas: signal and image processing, speech (G.728, GSM-AMR, Echo Canceller), audio (MP3, AAC), and video (MPEG-2, MPEG-4, H.264, VC1) coding, image coding, data compression (BWT, MFT, RLE, LZSS, LZ77), cryptography (SHA, AES, RSA certified by NIST), text processing and computer vision. The Intel IPP software runs on different operating systems: Windows* OS, Linux* OS and Mac OS* X.

It may be useful to emphasize that the image processing operations in Intel® IPP library differ from the operations Intel® MKL library expose to operate on matrixes. Both the libraries operate on 2D data. The difference is in data type: Intel® IPP deals with images, which can be of integer and floating point data type and be 1, 3 and 4-channel images; Intel® MKL operates on matrixes of floating point data, real and complex.

Intel® IPP is a C-style API library. However, due to the stdcall calling convention, the primitives can be used in applications written in many other languages. For example, they work in applications written in Fortran, Java*, Visual Basic, C++, and C#. The Intel® IPP functions can be used in the Microsoft .NET Framework managed environment to speed up the performance of applications on Intel processors and compatible platforms.
C# interface to Intel® IPP. Namespace and Structures

The Intel® IPP software additionally to the libraries provides special wrapper classes for the functions of several Intel® IPP domains: signal and image processing, color conversion, cryptography, string processing, data compression, jpeg coding and math and vector math. The wrappers allow Intel® IPP users to call Intel® IPP functions in C# applications. These classes, for example, for the signal sp and image processing ip functions, are declared as follows:

```csharp
// ipps.cs
namespace ipp {
    public class sp {
        ...
    }
};
// ippi.cs
namespace ipp {
    public class ip {
        ...
    }
};
```

Enumerated data types that are used in the Intel® IPP library must be declared in the wrapper classes with a keyword `enum`. For example, the enumerator `IppRoundMode` is declared in `ippdefs.cs` as follows:

```csharp
public enum IppRoundMode {
    ippRndZero = 0,
    ippRndNear = 1,
    ippRndFinancial = 2,
};
```

Many Intel® IPP functions use structures as parameters. To pass structures from a managed environment to an unmanaged environment, the managed class `struct` must comply with the corresponding library in the unmanaged code. The attribute `StructLayout` is used with the value `LayoutKind.Sequential`. The code below shows how to use the structure `Ipp64fc`, which is type of a double complex number:

```csharp
[StructLayout(LayoutKind.Sequential, CharSet=CharSet.Ansi)]
public struct Ipp64fc {
    public double re;
    public double im;
    public Ipp64fc( double re, double im ) {
        this.re = re;
        this.im = im;
    }
};
```

Almost all Intel® IPP functions have pointers as parameters. The functions with pointers must be declared in the unsafe context. For example, the function `ippiAndC_8u_C1R` is declared as:
using System;
using System.Runtime.InteropServices;
namespace ipp {
    public enum IppStatus {
        ... ippStsNoErr = 0,
        ...
    };
    [StructLayout(LayoutKind.Sequential, CharSet=CharSet.Ansi)]
    public struct IppiSize {
        public int width;
        public int height;
        public IppiSize( int width, int height ) {
            this.width = width;
            this.height = height;
        };
    };
    unsafe public class ip {
        [DllImport("ippi-6.1.dll")]
        public static extern IppStatus ippiAndC_8u_C1R( byte* pSrc, int srcStep, byte value,
                                                    byte* pDst, int dstStep, IppiSize roiSize );
    };
}

This code must be compiled with the /unsafe option.

Note that working with pointers in the C# application requires using the operator fixed. The fixed statement sets a pointer to a managed variable, and this variable is used during execution of the statement. Without the fixed statement, pointers to managed variables may be relocated unpredictably by the garbage collector.

For the class Bitmap, the methods LockBits, UnlockBits must be used.

**Intel® IPP Components - Image Processing Sample**

Intel® IPP Samples include C# interface and a demo application demonstrating how C# developers can build applications with Intel® IPP calls [4]. The demo application performs filtering, morphological and geometric operations. Additionally the image compression functions are used in the demo to read and write JPEG files. The application uses the wrapper classes for the image processing (ippi.cs) and image compression (ippj.cs) domains. The application launches the P/Invoke mechanism for unmanaged code in ippi-6.1.dll and loads the dispatcher of the processor-specific libraries. This dispatcher loads the most efficient library for a given processor, for example, the library ippiv8-6.1.dll is loaded on a system with Intel® Core™ 2 Duo processor, and the library ippiw7-6.1.dll is loaded on a system with Intel®Pentium® 4 processor.
Image ROI Processing

Special attention must be paid when working with the functions that require border pixels, for example, image filtering functions. The Intel® IPP functions operate only on pixels that are part of the image. Therefore a region of interest (ROI) is implied to be inside the image in such a way that all neighborhood pixels necessary for processing the ROI edges actually exist.

For example, for filtering functions the width of the border outside ROI must not be less than half of the filter kernel size with the centered anchor cell.

When processing an image ROI, the developer has to perform two additional operations: shifting the pointer to the data and specifying the ROI size that is less than the image size. The following code illustrates how to work with ROI (using the example of the Intel IPP function ippiFilterBox that performs image blurring):

```csharp
namespace ExampleIP
{
    using System;
    using System.Windows.Forms;
    using System.Drawing;
    using System.Drawing.Imaging;
    using ipp;

    public class tip : System.Windows.Forms.Form {
        private System.Drawing.Bitmap bmpsrc, bmpdst;

        private BitmapData getBmpData( Bitmap bmp ) {
            return bmp.LockBits( new Rectangle(0,0,bmp.Width,bmp.Height),
                ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb );
        }

        unsafe private void FilterBoxFunction() {
            BitmapData bmpsrcdata = getBmpData( bmpsrc );
            BitmapData bmpdstdata = getBmpData( bmpdst );
            IppiSize roi = new IppiSize( bmpsrc.Width*3/4, bmpsrc.Height*3/4 );
            const int ksize = 5, half = ksize/2;
            // the three-channels images
            byte* pSrc = (byte*)bmpsrcdata.Scan0+(bmpsrcdata.Stride+3)*half,
                pDst = (byte*)bmpdstdata.Scan0+(bmpdstdata.Stride+3)*half;
            IppStatus st = ipippiFilterBox_8u_C3R(pSrc,bmpsrcdata.Stride,
                pDst,bmpdstdata.Stride,
                roi,
                new IppiSize(ksize,ksize),
                new IppiPoint(half,half)));

            ...
        }
    }
}
```
Runtime Function Invocation

The above code example shows how an Intel IPP function is launched via a direct call, and static code is generated when the application is compiled. This method is rather simple and obvious.

When calling functions with the same set of parameters, you can use the dynamic method of function search and execution. This method is called reflection and can noticeably reduce the size of the executable code.
In the demo application the reflection method is used to launch filtering and morphological functions from the menu as follows:

```csharp
namespace ExampleIP
{
    using System;
    using System.Windows.Forms;
    using System.Drawing;
    using System.Drawing.Imaging;
    using System.Reflection;
    using System.Collections;
    using ipp;

    public class tip : System.Windows.Forms.Form
    {
        private Assembly assembly;
        private object ippi;
        private Type ippiType;
        private Hashtable hash;

        public tip(Bitmap bmp)
        {
            assembly = Assembly.LoadFrom("ippi_cs.dll");
            ippi = assembly.CreateInstance("ipp.ip");
            ippiType = ippi.GetType();

            assembly = Assembly.LoadFrom("ippi_cs.dll");
            ippi = assembly.CreateInstance("ipp.ip");
            ippiType = ippi.GetType();

            public tip(Bitmap bmp) {
                assembly = Assembly.LoadFrom("ippi_cs.dll");
                ippi = assembly.CreateInstance("ipp.ip");
                ippiType = ippi.GetType();
            }
            void CreateMenu()
            {
                hash = new Hashtable();
                MenuItem miBlur = new MenuItem("Blur", new EventHandler(MenuFilteringOnClick));
                hash.Add(miBlur, "ippiFilterBox_8u_C3R");
                MenuItem miMin = new MenuItem("Min Filter",
                new EventHandler(MenuFilteringOnClick));
                hash.Add(miMin, "ippiFilterMin_8u_C3R");

                hash = new Hashtable();
                MenuItem miBlur = new MenuItem("Blur", new EventHandler(MenuFilteringOnClick));
                hash.Add(miBlur, "ippiFilterBox_8u_C3R");
                MenuItem miMin = new MenuItem("Min Filter",
                new EventHandler(MenuFilteringOnClick));
                hash.Add(miMin, "ippiFilterMin_8u_C3R");

                hash.Add(miMin, "ippiFilterMin_8u_C3R");
            }

            private void MenuFilteringOnClick(object sender, System.EventArgs e) {
                FilteringFunction((string)hash[sender]);
            }

            unsafe private void FilteringFunction(string func) {
                MethodInfo method = ippiType.GetMethod(func);
                const int ksize = 5, half = ksize / 2;
                byte* pSrc = (byte*)bmpsrcdata.Scan0 + (bmpsrcdata.Stride + 3) * half;
                byte* pDst = (byte*)bmpdstdata.Scan0 + (bmpdstdata.Stride + 3) * half;
                IppStatus st = (IppStatus)method.Invoke(null, new object[]
                {
                    (IntPtr)pSrc, bmpsrcdata.Stride, (IntPtr)pDst, bmpdstdata.Stride, roi, new IppiSize(ksize, ksize), new IppiPoint(half, half)});
            }
        }
    }
}
```
**Intel® MKL and Intel® IPP Performance from C#**

Intel® IPP and Intel® MKL libraries are optimized for the Intel® and compatible processors. This optimization can speed up performance of various applications. However specific features of using these libraries in the .NET Framework can decrease performance of the application. Consider two major possible reasons of such decrease:

- Managed C# code calls unmanaged code through P/Invoke. Every P/Invoke call requires from 8 to 27 CPU clocks.
- When a function is called from a DLL for the first time, the corresponding DLL must be loaded into memory (for example, using LoadLibrary() on Windows OSs), which takes 1000-2000 CPU clocks. More CPU clocks are needed to create the entry points for all functions that are exported by this DLL.

A table with the performance overhead numbers for Intel® MKL DGEMM functions and Intel® IPP image processing functions is given below. Because Intel® MKL performs a larger number computational operations the overhead of its function call is less compared to Intel® IPP. Because Intel® IPP functions are faster of corresponding C# implementations by several times, it still makes sense to call Intel® IPP in spite of the overhead of C# call. To decrease the overhead effect developers can create a component or application level interface in which one C# call leads to many IPP functions execution. An example of this approach is the .NET interface for DMIP described later in this paper. Also, we can compare performance of the C# implementation and C# call of Intel® IPP functions. For example, C# .NET library Mirror costs 5.7 CPU cycles per pixel, Intel® IPP based C# call of Mirror function costs 1.7 cycles per pixel.

<table>
<thead>
<tr>
<th>Intel® MKL function</th>
<th>Performance overhead</th>
<th>Intel® IPP function</th>
<th>Performance overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGEMM size 3-70</td>
<td>2-20%</td>
<td>ippiFilterMedian_8u_C3R</td>
<td>14%</td>
</tr>
<tr>
<td>DGEMM size &gt; 80</td>
<td>&lt; 1%</td>
<td>ippiRotateCenter_8u_C3R</td>
<td>14%</td>
</tr>
<tr>
<td>SDOT size 3-25</td>
<td>10-50%</td>
<td>ippiWarpAffine_8u_C3R</td>
<td>39%</td>
</tr>
<tr>
<td>SDOT size &gt; 30</td>
<td>&lt; 1%</td>
<td>ippiMirror_8u_C3R</td>
<td>47%</td>
</tr>
</tbody>
</table>

*Table 2. Cost of C# call. The lighter an operation, the bigger the overhead*

The performance gain provided by optimized Intel® IPP and Intel® MKL functions is so significant that the overhead of function calls may be considered negligible, especially for computed intensive operations (such as Fast Fourier transform) or operations on large datasets.

**Conclusion**

The managed-unmanaged code interoperability provided in the .NET Framework environment can be performed in different ways.

The best way to call C functions residing in a native DLL is the use of the P/Invoke service, which is available in any managed language. P/Invoke provides a powerful and flexible interoperability with inherited codes. The DllImport attribute declares the external entry point. Marshalling attributes
allow describing various options for the data conversion and data layout in addition to the default data marshalling. The use of SuppressUnmanagedCodeSecurityAttribute, InAttribute, and OutAttribute may significantly reduce the performance overhead. Automatic pinning of the objects passed prevents them from being garbage collected for the duration of the call. Manual pinning is also available if the pointer to the object is kept and used in native code after the call returns. A managed delegate type allows implementing callback functions. The performance overhead of .NET interfaces is low relative to the significant performance benefits the Intel® MKL and Intel® IPP libraries provide for the core computations.

Appendix A.

Deferred Mode Image Processing and C++ Interoperability

C++ Interoperability

C++ interoperability is the second way to call unmanaged code from managed code in the .NET Framework environment. C++ interoperability allows managed and unmanaged code to co-exist and interoperate within the same assembly. It supports superset of interoperate functionality whereas other .NET Framework languages, for example Visual Basic, supports only P/Invoke.

Mixed Native and Managed Assembly

The assemblies that contain both unmanaged code with machine instruction and CIL instructions are called mixed assemblies. They can call and can be called by the .NET Framework components, and at the same time they retain compatibility with the unmanaged components. The mixed assembly is an ideal way of migrating existing C++ applications to the .NET Framework.

P/Invoke and C++ Interoperability

P/Invoke is available in all .NET Framework languages. C++ interoperability is available only in C++. P/Invoke is not type-safe, whereas C++ interoperability is type-safe. It allows reporting errors at run time. The C++ interoperability mechanism also has some performance advantages over P/Invoke. Both P/Invoke and C++ interoperability require some actions when a managed function calls an unmanaged function:

- Marshalling: the function call arguments are marshaled from CLR to the native types
- Thunking: a managed-to-unmanaged thunk is executed
- Unmanaged call: the unmanaged function is called with the native version of the arguments
- An unmanaged-to-managed thunk is executed
- The return type and any output arguments are marshaled from the native types to the CLR types.

The managed/unmanaged thunking is always needed for interoperability. The need of data marshalling depends on the data types, function signature, and further use of data.
C++ interoperability uses the simplest possible form of data marshalling: the parameters are simply copied across the managed/unmanaged boundary without any transformation. In P/Invoke, it is possible only if all parameters are of isomorphic types. On the other hand, P/Invoke performs very robust steps converting each managed parameter to an appropriate native type, and vice versa for output parameters.

**.NET Wrappers for Native C++ DMIP/UIC High Level API**

DMIP stands for Deferred Mode Image Processing layer and UIC stands for Unified Image Codec interface. Those are both C++ high-level libraries built on top of Intel® IPP. These libraries provide domain specific high-level API that simplifies the use of Intel® IPP in the image processing specific tasks. The .NET Framework interface for these high-level Intel® IPP libraries is built with C++ interoperability. To do this special wrapper classes with the .NET Framework semantic are written to represent similar API but in managed environment. The wrapper classes are used to create mixed assembly to expose high-level API for .NET compliant languages while still based on native code. Figure 2 and pseudo-code snippets below show the example of the C++ class and appropriate .NET Framework wrapper. For simplicity not all methods of the class are shown. For complete source code of DMIP and UIC .NET Framework interface please refer to the Intel® IPP Samples package [3].

![Mixed assembly .NET DMIP DLL](image)

*Figure 2. IPP based DMIP in .NET environment.*
**Unmanaged C++ Class**

class DMAPI Image
{
  protected:
    void*      ptr;
    IppDataType type;
    IppChannels chan;
    IppiSize   roi;
    int        step;
  
  public:
    Image(void* data, IppDataType type, IppChannels chan, IppiSize roi, int step, int top=0, int bottom=0, int left=0, int right=0, bool inverserow=false);
    ~Image(void);
    
    virtual void* Data(void);
    virtual IppDataType Type(void);
    virtual IppChannels Channels(void);
    virtual IppiSize Roi(void);
    virtual int Step(void);
};
.NET Wrapper for the C++ Class

```csharp
using namespace Intel::ipp;
using namespace Intel::ipp::dmip;

public ref class Image
{
public:
    // constructor
    Image(IntPtr data, IppDataType type, IppChannels chan, IppiSize^ roi, int step, int top, int bottom, int left, int right, bool inverserow)
    {
        m_image = new DMIP::Image(data.ToPointer(),
            (::IppDataType)type,
            (::IppChannels)chan,
            (::IppiSize)roi,
            step, top, bottom, left, right, inverserow);
    }
~Image();

    property IppDataType Type { IppDataType get(); }
    property int Step { int get(); }
    property IppChannels Channels { IppChannels get(); }
    property IppiSize Roi { IppiSize get(); }

    operator DMIP::Image*() { return m_image; }
private:
    DMIP::Image* m_image;
};
```

Performance of DMIP Library

Compare the performance of the native C++ implementation of DMIP and the .NET Framework version of DMIP discussed above using an example of the image harmonization filter.

The DMIP operation graph for the harmonization filter is presented on Figure 3, where `Src` is a source image, `To32f` is image data conversion, in this example - unsigned char to single floating point data, `Sub` and `Mul` arithmetic operations on the image, and `To8u` is a final conversion data to original format - unsigned char.

![Figure 3. DMIP graph of the harmonization filter](image)

An example of the pseudo-code for the implementation of the harmonization filter with DMIP C++ at the symbolic level API looks like this:
The performance data (in cpx) measured for the harmonization filter implemented in native C++ code and in C# with use of DMIP .NET Framework wrappers is shown in the table below.

Measurements were performed on a 3.2 GHz Intel® Core™ i7 processor-based system with 4GB memory and 64-bit version of the Microsoft Windows Vista® OS.

<table>
<thead>
<tr>
<th>Image Size</th>
<th>C++</th>
<th>C#</th>
<th>Slice Size</th>
<th>C# / C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>512x512</td>
<td>20.2</td>
<td>23.8</td>
<td>192</td>
<td>1.2</td>
</tr>
<tr>
<td>1024x124</td>
<td>18.8</td>
<td>20.3</td>
<td>96</td>
<td>1.1</td>
</tr>
<tr>
<td>2048x2048</td>
<td>19.0</td>
<td>20.6</td>
<td>48</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Table 3. DMIP Performance of Harmonization filter with C++ and C# .NET interface

References


[3] Intel® IPP Samples


Using Intel® MKL and Intel® IPP in Microsoft® .NET® Framework

[15] DllImportAttribute Class
[16] Blittable and Non-Blittable Types
[22] cbrumme's WebLog. Asynchronous operations, pinning:
  http://blogs.msdn.com/cbrumme/archive/2003/05/06/51385.aspx
[23] Some WinAPI translates from C++ to C# : http://social.msdn.microsoft.com/Forums/en-US/csharpgeneral/thread/0ade294c-a1dc-4f85-9506-2a34b703e353/

Some additional useful links

Intel® MKL KB: http://software.intel.com/en-us/articles/intel-mkl-kb/all/1/
Intel® IPP KB: http://software.intel.com/en-us/articles/intel-ipp-kb/all/1/
Intel® IPP Demo : http://www.intel.com/software/products/Flash/ipp/SPD_nav.swf

Useful Articles

“Maximizing your image application performance for dual and multi-core systems”:  
http://www.advancedimagingpro.com/publication/article.jsp?pubId=1&id=3510

“H.264 and video compression”: http://www.ddj.com/201203492?pgno=1
Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101