Using SIMD Instructions in Windows* 8 Store Apps

Abstract
SIMD instruction sets can be used to boost performance in Windows 8 Store applications. This document focuses on how to create a SIMD library that can be consumed by any Windows 8 Store application programmed in any supported language. This paper shows how to begin making a SIMD library programmed in C++/CX, and three similar applications that use libraries written in C++/CX and XAML, C# and XAML, as well as JavaScript and HTML5. The SIMD library will aid in speeding up the applications.

Introduction
Windows 8 introduced a new interface called Windows 8 UI; it is a vast departure from traditional Windows UIs. Legacy applications cannot run on top of the new UI as application development is built on top of a new environment known as the Windows Runtime. We ask: will support for certain hardware features be available as they were for desktop development?

This article shows how SIMD can take advantage of hardware features like SSE and AVX, to boost performance in Windows Store applications. We will explain how to include SIMD features in C++, C#/VB, or JavaScript applications.

SIMD Instruction Sets
SIMD stands for Single Instruction Multiple Data. That’s what the instruction does - applies a single instruction or calculation to multiple pieces of data. Vectorizing a collection of data exposes application runtime parallelism. Applications with heavy loads of calculations that don’t take advantage of SIMD will use scalar calculations and produce one result for each instruction. A vectorized solution will perform the same calculation on a group of two or more pieces of data for each instruction. With SIMD to vectorize the same collection of data, applications have the ability to increase throughput.

Intel introduced three SIMD instruction families starting in 1996. The first was MMX (not implemented in sample). Intel® architecture allowed up to eight pieces of data to be calculated using one instruction. In 1999, SSE, Streaming SIMD Extension, was introduced and allowed 2x more throughput than MMX instructions when applied to the same application. This means SSE can perform calculations to 16 pieces of data. Four expansions were built on top of SSE to further its capabilities with SSE2, SSE3, SSSE3, and SSE4. Introduced in 2011, Intel® Advanced Vector Extensions (Intel® AVX) is a set of instructions that allow 2x more throughput than SSE and 4x more throughput than MMX in the same application. Intel® AVX capabilities allow it to perform calculations for up to 32 pieces of data.
The Sample Applications
We assume you understand how to build a Windows 8 Store app. The sample applications are built from the templates in Visual Studio 2012. There are references at the end of the document that show you how to build a Windows 8 Store application.

To get started, open any of the three sample applications in Visual Studio* 2012. Compile and run the program and you will see four sections. One section generates data and the other three execute calculations on the generated data. Every time the Data Size or Loop Count is changed, you must tap “Generate and Set Loop Count!” Tap any of the execution buttons, one at a time, and a progress bar will show that it is working.

Figure 1: Using the Application

Each sample will have the same UI, so take your pick. When running one of the applications, you will find that you can activate all three executions at once, but you will have to wait longer for all of them to finish.

Now let’s look at the code from SIMDLibrary. Open SIMDLibrary.sln in Visual Studio 2012. Inspect GenerateData.h and you will find the definition of the GenerateData class. The class defines asynchronous methods of each type of execution, scalar and vector calculations, because we know this kind of calculation will take over one second. Failing to make this asynchronous will cause any application that uses this library to block and prevent the user from continuing work on the app or having UI feedback on the screen.
Code 1: SIMD Library Class used by Sample Applications

```csharp
public ref class GenerateData sealed
{
    public:
    GenerateData();
    Windows::Foundation::IAsyncOperation<double>^ GenerateAsync(int32 numGenerate, int32 numLoop);
    Windows::Foundation::IAsyncOperation<double>^ RegExecuteAsync();
    Windows::Foundation::IAsyncOperation<double>^ SSEExecuteAsync();
    Windows::Foundation::IAsyncOperation<double>^ AVXExecuteAsync();

    private:
    float32* Data;
    uint32 DataSize;
    uint32 LoopCount;
};
```

Each calculation is wrapped in the method. Each method executes the same calculation, but is implemented differently depending on which calculation, scalar or vectorized, we are doing in SSE or AVX.

Inspect one of the execution methods and see how to construct an asynchronous object that can be used by Windows 8 Store applications.

Code 2: Asynchronous Object is returned to Caller

```csharp
IAasyncOperation<double>^ GenerateData::SSEExecuteAsync()
{
    return create_async( [this] () {
        time_t start, end;
        time(&start);
        for(uint32 i = 0; i < LoopCount; i++)
        {
            for(uint32 j = 0; j < DataSize; j += 4)
            {
                auto x = _mm_loadu_ps(&Data[j]);
                auto y = _mm_add_ps(x, x);
                auto z = _mm_mul_ps(x, x);
                z = _mm_add_ps(z, y);
                _mm_storeu_ps(&Data[j], z);
            }
        }
        time(&end);
        return difftime(end, start);
    });
}
```
The C++ syntax above may not seem familiar to you. It was introduced in C++11. A reference at the end of the document can show you how to write lambda functions. The C++ compiler has intrinsic functions that are recognized as SSE or AVX instructions. The other supported languages on the Windows RunTime do not recognize SEE or AVX intrinsic functions.

The create_async() function packages up a lambda function and returns an instance of the implementation to the caller. Then it’s ready to be executed whenever the caller sees fit. This model of asynchronous program is used the same way for each other asynchronous methods programmed in this library. C++, C#, and JavaScript have its own model of invoking asynchronous code.

After creating a library that executes SIMD instructions, you can build other applications that consume these components. For each new project using a SIMD library, we will want to reference it.

For each sample application, follow these directions:

1. Right click on your solution in the Solution Explorer tab
2. Choose Add
3. Choose Existing Project
4. Browse to find the SIMD library solution file to add the project to the application you’ll program

Figure 2: Adding Library to Current Project for Easy Reference
After these steps, you can create instances of the GenerateData class because you will have the application reference the SIMD Library.

1. Right click on the project
2. Click References
3. Click Add New Reference
4. On the left where Solution collapses is Projects, choose SIMDLibrary and hit OK

With the GenerateData class, you can now run SIMD instructions. The following code illustrates how each language instantiates the GenerateData class so the application can invoke asynchronous operations with SIMD instructions.

**Code 3: C++ Declaration of GenerateData within MainPage.xaml.h**

```c++
namespace SIMDSampleAppCpp
{
    public ref class MainPage sealed
    {
        ...
        private:
            //SIMD class which wraps all the SIMD operations
            SIMDLibrary::GenerateData^ Data;
        ...
    }
}
```

**Code 4: C++ Instantiation of GenerateData within MainPage.xaml.cpp at page construction**

```c++
MainPage::MainPage()
{
    Data = ref new GenerateData();
    InitializeComponent();
}
```

**Code 5: C# Declaration and Instantiation of GenerateData within MainPage.xaml.cs**

```csharp
public sealed partial class MainPage : SIMDSampleAppCS.Common.LayoutAwarePage{
    //SIMD class which wraps all the SIMD operations
    private GenerateData Data;

    public MainPage()
    {
        Data = new GenerateData();
        InitializeComponent();
    }
    ...
}
```
Code 6: JavaScript Declaration and Instantiation of GenerateData within default.js

```javascript
(function () {
    "use strict";
    ...
    // SIMD Class that wraps SIMD operations
    var SIMD = new SIMDLibrary.GenerateData();
    var dataSize = 0;
    var loopCount = 0;
    ...
})();
```

After construction of the GenerateData, each application can generate data to calculate with the scalar method and vectorized methods, SSE and AVX. Each language has their own model of invoking asynchronous objects. We’ll inspect the SSE asynchronous invocation to run SIMD instructions for each language below. Each other asynchronous method follows this same model.

Code 7: C++ Invocation of Asynchronous SIMD Method

```c++
void SIMDSampleAppCpp::MainPage::SSETapped(Platform::Object^ sender,
                                            Windows::UI::Xaml::Input::TappedRoutedEventArgs^ e)
{
    // UI feedback
    SSEProgBar->Visibility = static_cast<Windows::UI::Xaml::Visibility>(0);
    auto asyncOperation = Data->SSEExecuteAsync();
    SSETime->Text = "Working...";

    // Call async operation of SIMD class to execute SSE operations
    // Calculations is handled by the SIMD class and is not implemented here
    create_task(asyncOperation).then(
        [this] (double time)
        {
            // UI feedback
            SSETime->Text = time + " sec to execute";
            SSEProgBar->Visibility =
                static_cast<Windows::UI::Xaml::Visibility>(1);
        }
    );
}
```
Code 8: C# Invocation of Asynchronous SIMD Method

```csharp
private async void SSETapped(object sender, TappedRoutedEventArgs e)
{
    // UI Feedback
    SSEProgBar.Visibility = (Windows.UI.Xaml.Visibility)0;

    // Call async operation of SIMD class to execute SSE operations
    // Calculations is handled by the SIMD class and is not implemented here
    SSETime.Text = "Working...");
    var asyncOperation = Data.SSEExecuteAsync();
    await asyncOperation;

    // UI Feedback
    double time = asyncOperation.GetResults();
    SSETime.Text = time + " sec to execute";
    SSEProgBar.Visibility = (Windows.UI.Xaml.Visibility)1;
}
```

Code 9: JavaScript Invocation of Asynchronous SIMD Method

```javascript
function sseOnTapped() {

    // UI Feedback
    var resultText = document.getElementById("sseResults");
    var progressBar = document.getElementById("sseProgress");
    var progress = document.createElement("progress");
    progressBar.appendChild(progress);
    resultText.innerHTML = "Working...");

    // Call async operation of SIMD class to execute SSE operations
    // Calculations is handled by the SIMD class and is not implemented here
    SIMD.sseexecuteAsync().then(function (result) {

        // UI Feedback
        progressBar.removeChild(progress);
        resultText.innerHTML = result + " sec to execute";
    });
}
```

The languages model of invocation of asynchronous methods are very similar. They each receive the asynchronous object and invoke its code. Since it’s asynchronous, any code part of .then() method, await operator, or then function for C++, C#, or JavaScript respectively will be executed when the asynchronous code finishes.

The application does not handle any calculations. The SIMD library handles all calculations and hands off any values to the application after it finishes its job. Each application, with the use of the library, shows users how much faster vectorized data is compared to the computation of single pieces of data.
Performance Results

Figure 3 shows the results from the JavaScript version of the application. Regular, or scalar execution of the randomly generated data shows an execution time of around 231 seconds. As expected SSE and AVX execution on the same data shows enhanced performance. SSE finishes its execution 37 seconds and AVX finishes 30 seconds. This is almost 8x faster performance when using AVX. The C++ and C# version of the application produce similar results.

Figure 3: Results of Running JavaScript Sample Application

Conclusion

SIMD exposes vectorized calculations on workloads that require many calculations. This leads to performance boosts in applications. Windows 8 Store applications can take advantage of this technology. This technology can be exposed by programming a library of SIMD methods readily consumed by any Windows 8 Store application.
About the Author
My name is Maynard Gellada, but everyone calls me MJ. I’m an intern with Intel. During my internship I’ve had many opportunities to expand my knowledge by testing applications on different devices, programming a touch application, and developing web technologies. Born in Guam, I moved to a midsized city in Wisconsin when I was 7. I currently attend the University of Michigan, Ann Arbor where I am working towards my Bachelor of Science in Engineering for Computer Engineering. Along with my studies, I’m involved in the Asian American community, permitting me to hold many leadership positions. After graduation, I look forward to applying my engineering and interpersonal skills to this vast industry and challenge today’s problems.

References
Learn to build Windows Store Apps
Developing Windows Store Apps (C++/C#/VB and XAML)
Developing Windows Store Apps (JavaScript and HTML)
Lambda Functions in C++11 - the Definitive Guide
Creating Windows Runtime Components
Asynchronous programming
Creating Asynchronous Operations in C++ for Windows Store Apps
Asynchronous programming in C++
Asynchronous programming in .NET
Asynchronous programming in JavaScript
Notices

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.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS’ FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

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 go to: http://www.intel.com/design/literature.htm

Intel, the Intel logo, VTune, Cilk and Xeon are trademarks of Intel Corporation in the U.S. and other countries.

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

Copyright © 2012 Intel Corporation. All rights reserved.

Optimization Notice

Intel's compilers 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 SSE2, SSE3, and SSE3 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Performance Notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

For more complete information about performance and benchmark results, visit Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.