

Microsoft Accelerator v2 Programming Guide

Preview Draft #2 – Version 2.1 – March 23, 2011

Abstract

This paper is sample-oriented discussion of how to use Microsoft® Accelerator v2 from Microsoft Research to implement array-processing applications. It includes basic background on Accelerator and the Accelerator application programming interface (API), and detailed walkthroughs of five sample applications.

**Note:**

* Most resources discussed in this paper are provided with the Accelerator package. For a complete list of documents and software discussed, see “Resources” at the end of this document.
* For Accelerator updates and software availability news, see   
  http://research.microsoft.com/en-us/downloads/e45cf1d7-fd3d-4e03-a219-c0f3fca9881e/

In this guide

[Introduction](#_Toc264440799)

[The Accelerator Programming Pattern](#_Toc264440800)

[Data-Parallel Array Objects](#_Toc264440801)

[Accelerator Operations](#_Toc264440802)

[The Target API](#_Toc264440803)

[Error Handling](#_Toc264440804)

[Accelerator Programming Fundamentals](#_Toc264440805)

[How to Implement Accelerator Applications](#_Toc264440806)

[Resources](#_Toc264440807)

[Appendix A: Selected Method Descriptions](#_Toc264440808)

[Appendix B: Complete Code for Stack\_Async and Stack\_Async\_CPP](#_Toc264440809)

[Appendix C: Complete Code for BoxCar2](#_Toc264440810)

Disclaimer: This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft, DirectX, Visual Studio, Win32, and Windows are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

Document History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Change |  |  |  |
| April 14, 2009 | Preview draft v.0.9 | | | |
| November 10, 2009 | Preview content supporting the “parity” release of Accelerator v2.0 | | | |
| June 16, 2010 | Preview Draft #2 | | | |

Contents

[Introduction 4](#_Toc264440816)

[The Accelerator Programming Pattern 5](#_Toc264440817)

[Data-Parallel Array Objects 5](#_Toc264440818)

[How to Create Data-Parallel Array Objects 6](#_Toc264440819)

[Parameter objects 6](#_Toc264440820)

[Accelerator Operations 8](#_Toc264440821)

[Element-Wise Operations 9](#_Toc264440822)

[Mathematical Operations 10](#_Toc264440823)

[Logical Operations 11](#_Toc264440824)

[Other Element-Wise Operations 13](#_Toc264440825)

[Type-Conversion Operations 14](#_Toc264440826)

[Reduction Operations 14](#_Toc264440827)

[Transform Operations 15](#_Toc264440828)

[Linear Algebra Operations 18](#_Toc264440829)

[The Target API 18](#_Toc264440830)

[Object creation 18](#_Toc264440831)

[The Multicore Target 19](#_Toc264440832)

[The DirectX 9 Target 19](#_Toc264440833)

[Object Destruction 20](#_Toc264440834)

[Eval](#_Toc264440835)

[uation 20](#_Toc264440835)

[Asynchronous Evaluation with the .NET API 21](#_Toc264440836)

[Asynchronous Evaluation with the C++ API 22](#_Toc264440837)

[Cancel Asynchronous Evaluation 23](#_Toc264440838)

[Error Handling 23](#_Toc264440839)

[Accelerator Programming Fundamentals 24](#_Toc264440840)

[How to Install Accelerator 24](#_Toc264440841)

[.NET Applications 25](#_Toc264440842)

[C++ Applications 26](#_Toc264440843)

[How to Implement Accelerator Applications 27](#_Toc264440844)

[Sample Walkthrough: Game of Life 28](#_Toc264440845)

[The Game of Life 28](#_Toc264440846)

[The Life Sample 29](#_Toc264440847)

[Sample Walkthrough: Sliding-Window Filter 32](#_Toc264440848)

[The Shift Operation 33](#_Toc264440849)

[An Obvious (and Not Very Efficient) Sliding-Window Filter Implementation 34](#_Toc264440850)

[Boxcar: An Efficient Sliding-Window Filter Implementation 35](#_Toc264440851)

[Sample Walkthrough: Two-Dimensional Convolution 36](#_Toc264440852)

[Sample Walkthrough: Asynchronous Evaluation with the Managed Target API 39](#_Toc264440853)

[Sample Walkthrough: Asynchronous Evaluation with the C++ Target API 40](#_Toc264440854)

[Sample Walkthrough: A Sliding-Window Filter Implemented with Parameter Objects 41](#_Toc264440855)

[Resources 44](#_Toc264440856)

[Appendix A: Selected Method Descriptions 45](#_Toc264440857)

[AddDimension 45](#_Toc264440858)

[DropDimension 45](#_Toc264440859)

[Expand 46](#_Toc264440860)

[Gather 47](#_Toc264440861)

[InnerProduct 48](#_Toc264440862)

[OuterProduct 49](#_Toc264440863)

[Pad 50](#_Toc264440864)

[Replicate 50](#_Toc264440865)

[Rotate 51](#_Toc264440866)

[Section 51](#_Toc264440867)

[Shift and ShiftDefault 53](#_Toc264440868)

[Stretch 54](#_Toc264440869)

[Transpose 55](#_Toc264440870)

[Appendix B: Complete Code for Stack\_Async and Stack\_Async\_CPP 56](#_Toc264440871)

[Stack\_Async Sample Code 56](#_Toc264440872)

[Stack\_Async\_CPP Sample Code 57](#_Toc264440873)

[Appendix C: Complete Code for BoxCar2 59](#_Toc264440874)

# Introduction

The performance of client computers has steadily increased over the years, largely as a result of steadily increasing processor speed. However, the rate of increase in processor speed has slowed in recent years. Instead, OEMs are boosting performance by adding CPU cores to their client systems. Dual-core processors are now the norm on client systems, and mainstream client systems with as many as 16 cores are expected to be common within the next year or two.

Most modern client systems—even single-core systems—also have a graphics adapter with a graphics processor (GPU) and dedicated on-board graphics memory. GPUs developed since 2001 typically include multiple shader processors running in parallel, creating what is effectively a separate multicore processor on the graphics card. Although GPUs are designed specifically for graphics processing, they can be programmed to function as a general-purpose processor (GPGPU).

This shift from faster processors to more processors creates challenges for application developers. You don’t have to rewrite an application to take advantage of faster processors—the application simply runs faster. However, if you install that application on a multiprocessor system, it probably won’t perform much better than on a single-processor system. To take full advantage of multiprocessor systems, you must usually rewrite the application using parallel-programming techniques.

Accelerator v2 provides an effective way for applications to implement array-processing operations using the parallel processing capabilities of multiprocessor computers. You don’t need to explicitly deal with the complications of parallelizing your application. Instead, you use the Accelerator application programming interface (API) to implement your array-processing code, and Accelerator handles all the details of parallelizing and running the computation. The Accelerator API is almost completely processor independent, so you can run the same array processing operations on any supported processor—including GPUs and multicore CPUs—with only minor changes.

This paper is sample-oriented discussion of how to use Accelerator to implement array-processing applications. The first part, “Accelerator Basics,” provides some basic background on Accelerator and the Accelerator API. The remaining sections are walkthroughs of several sample applications that demonstrate how to use Accelerator. There is also an appendix with a detailed discussion of the more sophisticated Accelerator operations, plus complete code for samples discussed in this paper.

**Important:** This paper assumes that you have already read “An Introduction to Microsoft Accelerator,” which provides essential conceptual background and a brief introduction to Accelerator programming.

# The Accelerator Programming Pattern

The basic framework for implementing array processing with Accelerator is:

1. Create input arrays.

2. Load each array from Step 1 into an Accelerator data-parallel array object.

3. Process the input data by applying Accelerator operations to the data-parallel array objects.

4. Evaluate the results of the operation on a target processor, which returns an array containing the processed data.

5. Free the target object.

The following sections describe the key elements of this pattern.

# Data-Parallel Array Objects

Accelerator does not operate directly on arrays. Instead, you package each array that you want to process as an Accelerator data-parallel array object and operate on the objects. A data-parallel array object—**BoolParallelArray**, **DoubleParallelArray**, **FloatParallelArray**, **IntParallelArray** or **Float4ParallelArray**—represents a one or two dimensional array of the associated primitive type: **bool**, **double**, **float**, **int**, or **Float4** values. **Float4** is an Accelerator structure that contains a quadruplet of **float** values, and is used primarily in graphics programming.

From an application perspective, a data-parallel array object is a largely opaque container for an array. You do not have direct access to the array elements, and cannot manipulate the array by index. However, all data-parallel array objects inherit from the **ParallelArray** type, which exposes several methods and properties that provide some basic information about the array.

Rank

A property that contains the rank of the associated array: 1 or 2. The C++ API exposes the equivalent **GetRank** method, which returns the rank.

Data-parallel objects are often referred to as rank 1 or 2, based on this property.

Dimensions

A method that returns a one or two element **int** array containing the dimensions of the array that the object represents.

For first rank objects, the array contains a single element that is set to the length of the array.

For second rank objects, the array contains two elements.

The first element is set to the number of rows and the second element is set to the number of columns. You use a similar array to specify array dimensions for many Accelerator operations.

GetLength

A method that returns the length of the array along a specified dimension.

Shape

For convenience and backward compatibility with Accelerator 1.1, the Accelerator .Net API also includes a **Shape** property, which is identical to the value returned by **Dimensions**.

## Parameter objects

Parameter objects—**BoolParallelArrayParam**, **DoubleParallelArrayParam**, **FloatParallelArrayParam**, **Float4ParallelArrayParam**, and **IntParallelArrayParam**. They improve efficiency and ease of programming by allowing applications to apply the same series of operations to different data sets without rebuilding the expression graph. You use parameter objects in much the same way as data-parallel array objects.

To use parameter objects

1. Implement a series of Accelerator operations, just as you would with data-parallel objects.

Instead of data-parallel array objects, pass parameter objects of the appropriate type to those operations that take multiple input data objects. Accelerator creates data nodes for the parameter objects, but they have no associated data.

2. Load the first set of data arrays into data-parallel array objects.

3. Before evaluation, call each parameter object’s **Bind** method to bind the appropriate data-parallel object to the parameter object.

Accelerator attaches the data-parallel array objects to the corresponding data nodes in the expression graph. You can also bind parameter objects to a constant value of the appropriate type. This is equivalent to binding the parameter object to a data-parallel array object that represents an array with all elements set to the specified value.

4. Call **ToArray**.

5. Repeat steps 2‑4 with the each remaining data set.

Accelerator simply attaches the new data to the same expression graph.

For example, assume that you want apply the same processing to a series of input arrays. With data-parallel array objects only, the implementation would look something like:

for (i = 1; i < numArrays; i++)

{

fpInput = new FPA(inputArrays[i]);

fpResult1 = PA.*Operation1*(fpInput, ...);

fpResult2 = PA.*Operation2*(fpResult1, ...);

...

fpFinalResult = PA.*OperationN*(fpResultN-1, ...);

resultArray = evalTarget.ToArray1D(fpFinalResult);

}

Accelerator must rebuild the expression graph for each input array. The following example implements the same operations by using parameter objects.

fpInput = new FloatParallelArrayParams( );

fpResult1 = PA.*Operation1*(fpInput, ...);

fpResult2 = PA.*Operation2*(fpResult1, ...);

...

fpFinalResult = PA.*OperationN*(fpResultN-1, ...);

for (i = 1; i < numArrays; i++)

{

fpInput.Bind(inputArrays[i]);

resultArray = evalTarget.ToArray1D(fpFinalResult);

}

Accelerator builds the expression graph once, and then attaches each input array to the *fpInput* data node.

Parameter objects combined with asynchronous evaluation provide a very powerful and flexible way to evaluate Accelerator operations. For an example, see “Sample Walkthrough: A Sliding-Window Filter Implemented with Parameter Objects” later in this paper.

**Important:** Accelerator operations must have input objects with compatible types and dimensions—usually the same type and dimensions. Be careful to bind parameter objects only to compatible data-parallel objects. Otherwise, Accelerator throws an exception.

Accelerator automatically checks parameter objects for type mismatches. Otherwise, the target is responsible for verifying that the graph is valid, and any related exceptions are thrown after you call **ToArray**.

For parameter objects, the two most common types of graph corruption are:

* Failure to bind one or more parameter objects to valid data-parallel array objects.
* Binding a parameter object to a data-parallel array object whose dimensions are incompatible with other objects in the graph.

In both cases, the target throws an **AcceleratorException**. The information included in the object depends on the particular target. For more discussion of **AcceleratorException** and how to handle it, see “Error Handling” later in this paper.

The second scenario can be particularly troublesome, because the dimension mismatch might not be immediately apparent to the target. For example, **Shift** operations or multiplying the object by a constant won’t cause any problems. However, if you eventually try to do something such as add the result of these operations to a correctly dimensioned object, the dimension mismatch becomes apparent and the target throws an exception.

This scenario is often difficult to debug, because the target can usually identify only the operation node where the problem surfaced, not the data node that is the source of the problem. You are responsible for determining where the buggy data was introduced, which can be difficult if that data node is well-removed from the operation.

For an example of how to use parameter objects, see “Sample Walkthrough: A Sliding-Window Filter Implemented with Parameter Objects” later in this paper.

## How to Create Data-Parallel Array Objects

There are two ways to create data-parallel array objects:

* Load a data array into an object.
* Create an object that represents an array of specified constants.

To load a data array into an object, pass the array to the constructor when you create a new instance of the data-parallel array class. The classes have constructors to handle one and two dimensional arrays of the appropriate type.

The following example uses the .NET API load a one-dimensional float array, *inputArray1*, into a **FloatParallelArray** object.

FloatParallelArray fpInput1 = new FloatParallelArray(inputArray1)

The equivalent C++ code is similar, but you must also specify the array dimensions.

FloatParallelArray fpInput1 = FloatParallelArray(inputArray1, dimensions);

For this example, *dimensions* is a **size\_t** value set to the array length. For a two-dimensional array, the second argument is a standard dimensions array with the first element set to the number of rows and the second element set to the number of columns. This array is used by a variety of Accelerator methods.

You can also create a data-parallel array object that represents a one or two-dimensional array with all elements set to a specified constant value. To do so, create a new object of the appropriate type and pass the constructor the constant value and the array dimensions.

The following example shows how to use the .NET API to create a new **FloatParallelArray** object that represents an array populated with 0s. For a first-rank object, set the second parameter to the length of the array. For a second rank object, set the second parameter to a dimensions array set to the array dimensions.

FloatParallelArray sum = new FloatParallelArray (0f, dimensions);

The following example shows the equivalent C++ code. The final parameter specifies the array’s rank.

FloatParallelArray sum = FloatParallelArray (0.0, dimensions, 2);

For a discussion of how to use such data-parallel array objects, see “Sample Walkthrough: Game of Life” later in this paper.

# Accelerator Operations

The Accelerator API includes a large collection of operations that applications can use to manipulate the contents of arrays and to combine arrays in various ways. Most operations take one or more input data-parallel array objects, and return a result data parallel array object that represents the processed arrays.

Some binary operations can take a constant in place of one of the input data-parallel array objects. In that case, the operation treats the constant as a data-parallel array object representing an array with all elements set to the constant’s value. The array dimensions are set to the dimensions of the other input object, which must be a data-parallel array object.

For example, the element-wise addition operator, **Add**, can take two data-parallel array objects as input, or one object and one constant. The following example shows the operation with two input data-parallel array objects, *parallelA* and *parallelB*.

parallelC = Add(parallelA,parallelB);

Each element of the result array, *parallelC*, is the sum of the corresponding elements of the two input arrays.

The following example shows an example of how to use **Add** with a constant input:

parallelC = Add(parallelA, 5);

In this case, each element of *parallelC* is the sum of the corresponding element of *parallelA*, and the specified constant, 5.

Accelerator exposes operations as a C++ functions and C# methods. The operation names are the same for both APIs and each operation has multiple overloads to handle the various input types. Some operations—including many of the element-wise operations—are also exposed as operators. **Add**, for example is also exposed as a ‘+’ operator.

The .NET API implements operations as static methods in the **Microsoft.ParallelArrays.ParallelArrays** class, which is exposed by Microsoft.Accelerator.dll. The C++ Accelerator operations are implemented as standard functions in the **ParallelArrays** namespace and are exported by name from Accelerator.dll. The associated header file is Accelerator.h. For simplicity, this paper focuses primarily on the .NET API, but the C++ syntax and usage are very similar.

The following sections provide brief descriptions of the commonly used Accelerator operations. For brevity, the examples use rank 1 objects, but most operations can also be used with rank 2 objects. For brevity, discussion assumes the following standard aliases for the data-parallel array objects.

using BPA = Microsoft.ParallelArrays.BoolParallelArray;

using DPA = Microsoft.ParallelArrays.DoubleParallelArray;

using FPA = Microsoft.ParallelArrays.FloatParallelArray;

using IPA = Microsoft.ParallelArrays.IntParallelArray;

using F4PA = Microsoft.ParallelArrays.Float4ParallelArray;

The corresponding parameter object aliases are BPAP, and so on.

## Element-Wise Operations

Element-wise operations operate separately on each element in the input object or objects. The element-wise operations are implemented as static methods in the ParallelArrays class. Some also have corresponding operator overloads, which are implemented by the data-parallel array objects that support the operation. For example, the & operation—which is equivalent to ParallelArrays.And—is implemented by the BoolParallelArray class. The \* operation—which is equivalent to ParallelArrays.Multiply—is implemented by the **DoubleParallelArray**, Float4ParallelArray, FloatParallelArray, and IntParallelArray classes.

The following sections describe the available element-wise operations, broken down by category.

### Mathematical Operations

Mathematical operations can be grouped into three categories: unary, binary, and trinary. Some operations can be used with any of the three numerical data-parallel array objects, but many apply only to **FloatParallelArray** and **Float4ParallelArray** objects. Tables 1 and 2 summarize the mathematical element-wise operations, including the supported object types.

#### Unary Operations

Unary operations take a single data-parallel array object, perform the specified operation on each element, and return the results in a new data-parallel array of the same type and dimensions.

Table 1. Unary Mathematical Operations

|  |  |  |
| --- | --- | --- |
| Method | Type | Description |
| Abs | DPA, FPA, F4PA, IPA | Determines the absolute value of each element. |
| Ceiling | DPA, FPA, F4PA | Determines the smallest integer that is greater than or equal to each element’s value. For example, if an element is set to 2.4, the corresponding element in the returned object is 3.0. |
| Cos | DPA, FPA, F4PA | Determines the cosine of each element. |
| Floor | DPA,FPA , F4PA | Determines the largest integer that is less than or equal to each element’s value. For example, if an element is set to 2.4, the corresponding element in the returned object is 2.0. |
| Fraction | DPA, FPA, F4PA | Determines the fractional part of each element. For example, if an element is set to 2.4, Fraction returns 0.4. |
| Log2 | DPA, FPA, F4PA | Determines the base 2 logarithm of each element. |
| Negate | DPA, FPA, F4PA, IPA | Determines each element multiplied by -1. |
| Pow2 | DPA, FPA, F4PA, IPA | Determines the value of 2 raised to the power of each element. |
| Reciprocal | DPA, FPA , F4PA | Determines the reciprocal of each element. |
| Rsqrt | DPA, FPA, F4PA | Determines the reciprocal of the square root of each element (1/sqrt(x)). |
| Sin | DPA, FPA, F4PA | Determines the sine of each element. |
| Sqrt | DPA, FPA, F4PA | Determines the square root of each element. |

#### Binary and Trinary Operations

Binary and trinary operations take two or three data-parallel array objects of the same type and dimensions. The operations act on the corresponding elements from each object and return the results in a new data-parallel array of the same type and dimensions. All input objects must have the same dimensions. Most of these operations also allow you to use a constant in place of one of the input objects.

Table 2. Binary and Trinary Mathematical Operations

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Type | Operator | Returns |
| Add | DPA, FPA, F4PA, IPA | + | Determines the sum of each pair of elements: (a1+b1, a2+b2, ...). |
| Subtract | DPA, FPA, F4PA, IPA | - | Determines difference between each pair of elements: (a1-b1, a2-b2, ...). |
| Multiply | DPA, FPA, F4PA, IPA | \* | Determines the product of each pair of elements: (a1\*b1, a2\*b2, ...). |
| Divide | DPA, FPA, F4PA, IPA | / | Determines the ratio of each pair of elements: (a1/b1, a2/b2, ...). |
| Min | DPA, FPA, F4PA, IPA | N/A | Determines the minimum value of each pair of elements. |
| Max | DPA, FPA, F4PA, IPA | N/A | Determines the maximum value of each pair of elements. |
| MultiplyAdd | DPA, FPA, F4PA, IPA | N/A | A trinary operation that determines the sum of the third of each set of elements and the product of the first two elements of the set: ((a1\*b1)+c1, (a2\*b2)+c2, ...). |
| Pow | DPA, FPA, F4PA, IPA | \*\* | Determines the value of the first of each pair of elements raised to the power of the second element of the pair: (a1\*\*b1, a2\*\*b2, ...). |

### Logical Operations

Logical operations can be grouped into three categories: Boolean, comparison, and selection. For detailed descriptions, see the reference pages in the Accelerator help file.

#### Boolean Operations

Boolean operations take one or two BoolParallelArray objects, perform a Boolean operation on the corresponding elements, and return the results in a new BoolParallelArray object. For **And** and **Or**, you can also set one of the input values to **true** or **false**.

Table 3. Boolean Operations

|  |  |  |
| --- | --- | --- |
| Method | Operator | Description |
| Not | ! | A unary operation that switches true elements to false and vice versa: (!a1, !a2, ...). |
| And | & | A binary operation that determines the result of a logical And on each pair of elements: (a1&b1, a2&b2, ...). |
| Or | | | A binary operation that determines the result of a logical Or on each pair of elements: (a1|b1, a2|b2, ...). |

#### Comparison Operations

Comparison operations perform an element-wise numerical comparison of a pair of **DoubleParallelArray**, FloatParallelArray, Float4ParallelArray, or IntParallelArray objects and return the result in a new BoolParallelArray object. You cannot use a constant as input. If the comparison of two elements evaluates to **true**, the corresponding element of the returned object is set to true. Otherwise, the element is set to false.

Table 4. Comparison Operations

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Type | Operator | Descriptions |
| CompareEqual | FPA, DPA, FPA | N/A | Determines whether each pair of elements are equal (a1==b1, a2==b2, ...). |
| CompareGreater | FPA, DPA, FPA | > | Determines whether the first element of the pair is greater than the second (a1>b1, a2>b2, ...). |
| CompareGreaterEqual | FPA, DPA, FPA | >= | Determines whether the first element of the pair is greater than or equal to the second (a1>=b1, a2>=b2, ...). |
| CompareLess | FPA, DPA, FPA | < | Determines whether the first element of the pair is less than the second (a1<b1, a2<b2, ...). |
| CompareLessEqual | FPA, DPA, FPA | <= | Determines whether first element of the pair is less than or equal to the second (a1<=b1, a2<=b2, ...). |
| CompareNotEqual | FPA, DPA, FPA | N/A | Determines whether each element of the pair is not equal to the second (a1 != b1, a2!=b2, ...). |

#### Selection Operations

Cond and Select are trinary operations that take three data-parallel array objects and return a new object with the same dimensions. Each element of the returned object is set to the corresponding element of the second or third object, based on the values of the corresponding element in the first object.

The Cond method takes a **BoolParallelArray** object followed by two **IntParallelArray**, **FloatParallelArray**, or **DoubleParallelArray** objects. You cannot use constants as inputs and all three objects must have the same dimensions. Cond returns an object of the same type and dimensions as the last two objects, as follows:

* If an element in the **BoolParallelArray** is set to true, Cond returns the corresponding element from the second object.
* If an element in the **BoolParallelArray** is set to false, Cond returns the corresponding element from the third object.

For example, assume that the input data-parallel objects represent the following arrays:

In this example, Cond(a, b, c) yields:

The Select method takes three objects that can be a combination of **DoubleParallelArray**, **FloatParallelArray**, **Float4ParallelArray**, or **IntParallelArray** objects or **double**, float or int constants. For a complete list of the possible combinations, see the reference pages in the Accelerator help file.

Select returns an object of the same dimensions and type as the first object, as follows:

* If an element in the first object is greater than or equal to zero, Select returns the corresponding element from the second object.
* If an element in the first object is less than zero, Select returns the corresponding element from the third object.

For example, assume that the input data-parallel objects represent the following arrays:

In this example, Select(a, b, c) yields:

### Other Element-Wise Operations

The only operation in this category is Interpolate, which takes three input **FloatParallelArray**, **DoubleParallelArray,** **Float4ParallelArray** objects, which must all have the same type and dimensions. The result object has the same type and dimensions as the input objects, and contains an interpolated value for each pair of elements from the last two objects. The elements in the first input object are set to values between zero and one, and specify how to weight the second element relative to the third when performing the interpolation.

For objects named a, b, and c, **Interpolate** uses the following formula—or the two-dimensional equivalent—to calculate the result object, x:

xi = ai\*bi + (1-ai)\*ci.

For example, assume that the input objects represent the following arrays:

In this example, Interpolate(a, b, c) yields:

### Type-Conversion Operations

Type conversion operations take a data-parallel array object and return another data-parallel object of the same dimensions but a different type. Table 5 contains the data-parallel array type conversion operations.

Table 5. Conversion to Data-Parallel Array

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Input Type | Output Type | Description |
| ToIntParallelArray | FPA | IPA | Converts an FPA or DPA to an IPA by truncating the fractional part of each floating point value. |
| ToDoubleParallelArray | FPA | DPA |  |
| ToFloatParallelArray | IPA, DPA | FPA | Converts an IPA or DPA to an FPA. |
| ToFloat4ParallelArray | FPA | F4PA | Converts an FPA to a F4PA in one of two ways:   * Pass four FPA objects to specify the individual members of the corresponding Float4 structure. * Pass a single FPA object to sets all four members of the corresponding Float4 structure to the same value. |
| FromFloat4ParallelArray | F4PA | FPA | Converts an F4PA to an FPA. The second parameter is an integer value from 1–4, which specifies which of the four elements of the Float4 structure to assign to the corresponding FPA element. |

## Reduction Operations

Reduction operations reduce the rank of a data-parallel array object in either of the following ways:

* Reduction across one dimension converts the object into another object of the same type, but with the rank reduced by one.
* Reduction across all dimensions converts the object into a first rank object of the same type with a single element, sometimes referred as a scalar object.

For example, assume that a is an IntParallelArray object that represents the following second rank array:

The Sum operation reduces rank by summing the elements of an array. For example, Sum(a,0) sums along each row and returns a first rank IntParallelArray object that contains the sum of each row:

Sum(a) sums every element in the array and produces IntParallelArray object with a single element set to 20.

Table 6 contains brief descriptions of the reduction operations.

Table 6. Reduction Operations

|  |  |  |
| --- | --- | --- |
| Method | Input Type | Description |
| Sum | DPA, IPA, FPA, F4PA | Reduces rank by summing the elements across a specified dimension or sums the entire array. |
| Product | DPA, IPA, FPA, F4PA | Reduces rank by taking the product of the elements across a specified dimension or the product of the entire array. |
| MaxVal | DPA, IPA, FPA, F4PA | Reduces rank by selecting the maximum values of the elements across a specified dimension or the maximum value for the entire array. |
| MinVal | DPA, IPA, FPA, F4PA | Reduces rank by selecting the minimum values of the elements across a specified dimension or the minimum value for the entire array. |
| Any | BPA | Reduces rank by performing a logical Or on the elements across the specified dimension or the result of a logical Or for all the elements in the array. |
| All | BPA | Reduces rank by performing a logical And on the elements across the specified dimension or the result of a logical And for all the elements in the array. |

## Transform Operations

Transform operations reorganize the elements of data-parallel array objects in various ways. They are sometimes referred to as memory transforms, because they move elements around without performing any computations. The operations fall into four basic categories:

* Expansion operations create a new and larger object based on the elements of the original object.
* Selection operations create a new object that contains a selected subset of the elements of the original object.
* Reordering operations reorganize the elements of the original object without changing the dimensions.
* Rank change operations change the rank of the original object.

Transform operations are used for many types of array processing tasks, including filtering digital streams and processing bitmaps. Most of the sample walkthroughs, later in this paper, use one or more transform operations.

If you visualize an array as a rectangle or window in memory, most Accelerator transforms can be thought of as Affine transforms, which are a class of linear transforms that includes translation, rotation, and scaling. In many cases, the transformed array contains elements that were not in the original array, so Accelerator must assign values to the unspecified elements. For example, start with the following 3x3 array:

The Shift transform translates the memory window in a specified direction. If you shift the memory rectangle one to the right—which shifts the array elements one to the left—the transformed array is:

where ‘\*’ indicates unspecified elements.

As another example, Expand adds rows or columns to the original array, which creates a row or column of unspecified values. The following example shows the result of adding a column to a:

Each transform operation that creates unspecified elements supports one of the following three options for assigning values to those elements. Using a as the original object, Accelerator assigns values to the unspecified elements in one of the following ways:

Replicate the closest elements.

For example, Shift assigns values to unspecified elements by replicating the closest elements from the original array. For a, replication yields:

Assign a specified constant value.

For example, ShiftDefault is similar to Shift, but assigns a specified constant value to the unspecified elements. For a, with 0 specified as the constant value, ShiftDefault yields:

“Wrap” the original array.

For example, Expand “wraps” the original array, which assigns the values from the opposite side of the array to the unspecified values. For a, this yields:

Table 7 provides a brief description of the Accelerator transforms. These operations can be performed on all types of data-parallel array objects, although some transforms are restricted to objects of a particular rank. For details on the individual operations, see Appendix A or the Accelerator help file.

Table 7. Transform Operations

|  |  |  |
| --- | --- | --- |
| Method | Type | Description |
| AddDimension | Rank Change | Transforms a first rank data-parallel array object into a second rank object by adding an empty dimension. |
| DropDimension | Rank Change | Transforms a second rank data-parallel array object into a first rank object by removing a dimension. The dimension to be removed must be empty. |
| Expand | Expansion | Increases the dimensions of a data-parallel array object and assigns values to the new elements by wrapping elements from the original object. |
| Gather | Selection | Creates a new a data-parallel array object that contains a specified subset of the elements of the original object. |
| Pad | Expansion | Increases the dimensions of a data-parallel array object and assigns a specified constant value to the new elements. |
| Replicate | Expansion | Increases the dimensions of a data-parallel array object, and assigns values to the new elements by putting the original elements in the upper left corner of the new array and using that array to tile the remainder of the array. |
| Rotate | Reordering | “Rotates” a data-parallel array object by cyclically permuting the rows or columns, but not changing the dimensions. |
| Section | Selection | Creates a subset of a data-parallel array object that consists of regularly spaced “slices” of the original array. |
| Shift/  ShiftDefault | Reordering | Translates the memory window represented by a data-parallel array object. The dimensions of the underlying array remain the same, but the elements are shifted left, right, up, or down relative to the original array. The two methods differ only in how they assign values to the unspecified elements that are created by the shift:   * Shift assigns the values of the closest elements from the original array to the unspecified elements. * ShiftDefault assigns a specified constant value to the unspecified elements. |
| Stretch | Expansion | Increases the dimensions of a data-parallel array object by replicating the existing elements a specified number of times. |
| Transpose | Reordering | Performs matrix transposition on a second rank data-parallel array object. |

## Linear Algebra Operations

Linear algebra operations take two data-parallel array objects and return a new object. Accelerator supports the following operations:

* Outer product
* Scalar product
* Matrix-vector multiplication
* Matrix-matrix multiplication

Table 8 provides a brief description of the linear algebra operations. For details on the individual operations, see Appendix A or the Accelerator help file.

Table 8. Linear Algebraic Operations

|  |  |  |
| --- | --- | --- |
| Method | Input Type | Description |
| InnerProduct | DPA, FPA, IPA | Performs scalar product, matrix-matrix multiplication, or vector-matrix multiplication, depending on the rank of the input objects. |
| OuterProduct | DPA, FPA, IPA | Creates a second rank object from the outer product of two first rank objects. Outer product is sometimes referred to as the tensor or dyadic product. |

# The Target API

When you apply Accelerator operations to data-parallel objects, Accelerator does not perform any computations. Instead, it constructs a directed acyclic graph (DAG) that represents the programming logic and data. The operations are executed only when you direct one of the Accelerator targets to evaluate a result data-parallel array object. The target uses the DAG that is associated with the result object to perform the calculation on the target processor and returns an array containing the processed data.

All Accelerator targets expose a standard C++ interface and most also expose an associated managed wrapper. Targets can optionally expose several additional features. For example, the DirectX® 9 target (DX9Target) exposes a target memory interface, which is discussed later in this section.

## Object creation

The mechanism for creating target objects depends on whether you are using the .NET or C++ API.

* Managed applications use the **new** operator to create a new target object.
* C++ applications call object-creation functions or static methods on the **Target** class, which return a pointer to the target object.

Target implementers can expose either functions or static methods. The only requirement for object creation function is that it return a new target object. The function’s name and syntax is determined by the target implementer and depends on the particular target. Both standard targets use functions. For other targets, see the associated documentation.

Targets can have multiple object-creation options, which allow you do configure the target object in different ways. The following sections describe object-creation options for the standard targets.

### The Multicore Target

Accelerator supports a multicore target for x64 CPUs, only. It supports both a C++ API—exposed by Accelerator.dll—and a managed API—exposed by Microsoft.Accelerator.dll. Both versions of the target class are named **X64MulticoreTarget**.

The multicore target supports several object creation options:

* By default, the target uses a standard resource manager, **SimpleAcceleratorResourceManager**, and runs on all cores with maximum precision.
* You can implement a custom resource manager and pass it to the target, which will use it in place of the standard resource manager.
* You can specify a “less precise” configuration, which runs evaluations more quickly but with lower precision.
* You can limit the number of cores that the target uses to run evaluations.

This option is intended primarily for debugging purposes. If you specify more than the available number of cores, the target throws an exception.

Custom resource managers must be exposed to the target in the same way as **SimpleAcceleratorResourceManager**. However, custom managers do not have to be able to limit the number of cores.

### The DirectX 9 Target

Accelerator supports both x86 and x64 DirectX 9 target objects, which can be used with any video adaptor that supports DirectX 9. It supports both a C++ API—exposed by Accelerator.dll—and a managed API—exposed by Microsoft.Accelerator.dll. All versions of the target class are named **DX9Target**.

The DirectX 9 target supports three object creation options. One creates a DirectX 9 target for the default video adaptor and the other two allow you to specify a particular video adaptor. For details on how to specify a video adaptor, see “DirectX Developer Center” on the MSDN Web site.

* With one video adaptor, you typically use the default option.
* With multiple video adaptors, you can use one of the latter two options to create a target object for each adaptor. You can then run multiple evaluations concurrently on different adaptors.

The .NET API exposes the **DX9Target** creation options through a set of constructors. The C++ API has an equivalent set of object creation functions, named **CreateDX9Target**.

## Object Destruction

Target objects require substantial resources, so you should free them as soon as the evaluation completes. You should also free target objects after the target throws an exception.

To free a target object

* C++ applications call **Target->Delete**.
* Managed applications call **Target.Dispose**.

## Evaluation

The bulk of the target API supports evaluation. The primary evaluation method is named **ToArray**, and has multiple overloads to accommodate the various data-parallel object types and ranks. **ToArray** methods take a data-parallel array object, and return an array of the appropriate type and rank through an **out** parameter.

For convenience, the .NET API supports a set of overloads named **ToArray1D** and **ToArray2D**. These methods are wrappers for **ToArray**, and return the result array as a return value instead of as an **out** parameter.

If a processor cannot support a particular object type, the target might provide only minimal implementations of some of the **ToArray** overloads, which typically throw an exception if you attempt to call them. For example, DirectX 9 does not provide native integer support, so the DirectX 9 target does not support **ToArray** overloads that evaluate **IntParallelArray** objects.

**Note:** The target API also supports a set of **ToBitMap** overloads, which are used to evaluate bitmaps. **ToBitmap** is not currently supported by any target.

**ToArray** supports both synchronous and asynchronous evaluation.

* With synchronous evaluation, the target blocks until evaluation is complete, and then returns a valid result array.
* With asynchronous evaluation, the target initiates evaluation on a separate thread and returns immediately.

When evaluation completes, the target populates the result array with the processed data and notifies the application that the result array contains valid data.

The details depend on whether you use the .NET or C++ API. The following sections discuss the details.

### Asynchronous Evaluation with the .NET API

The managed API uses the standard .NET **BeginInvoke**/**EndInvoke** pattern. Accelerator implements these methods as **BeginToArray** and **EndToArray**. To start an asynchronous evaluation, call the target’s **BeginToArray** method, which takes the following input:

* The data-parallel array object to be evaluated.

**BeginToArray** has overloads for the different data-parallel array object types and dimensions, much like **ToArray**.

* A result array to receive the processed data.

You can optionally use overloads that allow you to pass:

* A user-defined object.
* An **AsyncCallback** callback delegate, which is called when the computation is complete.

For more information on the **BeginInvoke**/**EndInvoke** pattern, see “Calling Synchronous Methods Asynchronously” in “Resources.”

**Note:** There is also a set of **ToArray** overloads that take an **ExecutionMode** value, which specifies normal, profile, or debug mode. This feature is not currently implemented.

**BeginToArray** starts the evaluation on a separate thread and immediately returns an **IAcceleratorAsyncResult** interface.

To start asynchronous evaluation

1. Call **BeginToArray** and pass in the data-parallel array object to be evaluated, and the result array.

**BeginToArray** immediately returns an **IAcceleratorAsyncResult** interface. Optionally, you can pass **BeginToArray** a user-defined object—typically the target object—and a callback delegate. They are used to handle completion notification.

2. Wait for the evaluation to complete.

To wait on the notification event, call the **IAcceleratorAsyncResult** object’s **WaitHandle.WaitAll** **WaitHandle.WaitAny**, or **WaitHandle.WaitOne** methods to wait on the completion notification.

Use a callback delegate, which is called by the target after the evaluation is complete.

The target notifies you when evaluation is complete. If you have provided a callback delegate, finalize the evaluation as follows.

To finalize asynchronous evaluation with a callback delegate

1. When the evaluation is complete, the target calls the callback delegate and passes it an **IAcceleratorAsyncResult** interface.

The user-defined object that you provided to **BeginToArray** is assigned to **IAcceleratorAsyncResult.AsyncState**, and is typically the target object.

2. Perform any necessary processing and return.

3. In the application proper, call **IAcceleratorAsyncResult.GetAcceleratorException** to verify that no exceptions were thrown during evaluation.

Because much of the asynchronous evaluation process takes place on a separate thread, you might not receive thrown exceptions. You must call **GetAcceleratorException** to be certain that no exceptions have been thrown.

4. If there are no exceptions, the result array contains valid data, which you can display, process further, and so on.

**Note:** You can call **EndToArray** at any time after calling **BeginToArray**. It is a synchronous method, and blocks until evaluation is complete and the result array contains valid data.

If you do not provide a callback delegate, the target simply signals the completion event. The target also signals the completion event after the callback delegate returns.

To finalize asynchronous evaluation with a completion event

1. Call **IAcceleratorAsyncResult.GetAcceleratorException** to verify that no exceptions were thrown during evaluation.

2. Exit the wait thread.

**CAUTION:** It is up to the target to decide when to populate the array with processed data. You must be certain that evaluation is complete, or **GetResult** might not return a valid array.

### Asynchronous Evaluation with the C++ API

The C++ API uses a set of **ToArray** overloads.

To perform asynchronous evaluation with the C++ API

1. Call **CreateEvent** to create an event handle.

2. To start evaluation, call the appropriate **ToArray** overload and pass it the event handle.

If you want to be able to cancel the evaluation, call a **ToArray** overload that returns an **IAcceleratorAsyncContext** interface

3. Wait on the event.

When evaluation is complete, the target populates the result array and signals the event to notify you that the result array is ready for use.

4. Call **IAcceleratorAsyncContext.GetAcceleratorException** to verify that no exceptions were thrown during evaluation.

If not, the result array contains valid data and you can proceed.

### Cancel Asynchronous Evaluation

You can cancel asynchronous evaluation at any time after starting. The target cancels the computation, cleans up its state, and then notifies the application using the same notification mechanism that it uses for successful completion.

**CAUTION:** Wait until the target notifies you that cancellation is complete before interacting with the result array. Otherwise, the target might still be interacting with the array, creating a risk of data corruption.

The details of implementing cancellation depend on the API.

To cancel an asynchronous evaluation with the managed API

1. Call **IAcceleratorAsyncResult.Cancel**.

2. If you intend to destroy the result array or dispose the target object, wait for a completion notification before proceeding.

If you provided a callback delegate, the target calls the callback delegate after cancellation is complete and signals the completion event after the delegate returns. Otherwise, the target just signals the completion event.

To cancel an asynchronous evaluation with the managed API

1. Call **IAcceleratorAsyncContext.Cancel**.

2. If you intend to destroy the result array or dispose the target object, wait on the completion event before proceeding.

## Sub expression Reuse

Unlike a compiler, Accelerator does not have the ability to see and analyze the entire program. Each call to ToArray is independent. If you have two Accelerator where there is a large sub expression that is used in both, Accelerator will evaluate the common sub expression in each call to ToArray.

For example

FPA a = some large expression

FPA b = A.Rotate(a \* 2.0f, new int[]{1, 1});

FPA c = A.Sin(a);

Result1 = evalTarget.ToArray2D(c);

Result2 = evalTarget.ToArray2D(b);

As always, the call to ToArray will evaluate the entire expression. The call has no memory of previous or knowledge of future invocations.

However, if you know that *a* is a large expression, it is possible to have Accelerator cache an intermediate result for a and reuse it.

FPA a = some large expression

FPA aCached = A.Evaluate(a);

FPA b = A.Rotate(aCached \* 2.0f, new int[]{1, 1});

FPA c = A.Sin(aCached);

Result1 = evalTarget.ToArray2D(c);

Result2 = evalTarget.ToArray2D(b);

By using an array that is the result of a call to Evaluate, such as *aChached* above, each target will store an intermediate value for that expression when it is first encountered. This stored value will be used all on all subsequent uses.

There is one caution. Creating and using an intermediate cached value reads and writes a great deal of memory. It is frequently the case that it is faster to allow the value to be recomputed. The use of Evaluate should be used judiciously.

# Error Handling

Both the Accelerator library and the various targets can throw exceptions, including:

* The Accelerator library can throw exceptions during graph construction for a variety of reasons.

For example, the library throws an exception if you attempt to add data-parallel array objects with different dimensions.

* The Accelerator library can throw an exception if you attempt to bind a data-parallel array object to a parameter object of a different type.
* Targets throw exceptions if you attempt to use an unsupported ToArray overload.
* After you have called **ToArray**, targets can throw exceptions if you bound an incorrectly sized data-parallel array object to a parameter object.

In general, you should wrap graph construction code, Bind, and ToArray calls in **try**-**catch** blocks.

Accelerator-specific issues, such as the ones in the preceding list, usually throw **AcceleratorException**, which contains the following data:

* A message string, which provides a readable description of the issue.
* A standard HRESULT value, usually E\_INVALIDARG or E\_OUTOFMEMORY.

The exception object is named **AcceleratorException** in both APIs, but the information is packaged somewhat differently. For the .NET API:

* Obtain the message string from the **AcceleratorException.Message** property.
* Obtain the HRESULT value from the **AcceleratorException.HResult** property.

For the C++ API:

* Obtain the message string by calling **AcceleratorException::GetReasonString**.
* Obtain the HRESULT value by calling **AcceleratorException::GetReason**.

# Accelerator Programming Fundamentals

This section discusses the basics of how to implement Accelerator applications. The most immediate question is whether to implement a .NET application or a C++ application. There is very little performance difference, so the choice is largely a matter of convenience.

* The .NET API is usually preferable for new applications, especially if you plan to implement a substantial user interface (UI).
* The C++ API is typically used to add Accelerator code to an existing C++ application.

For more discussion of this issue, see “An Introduction to Microsoft Accelerator.”

The remainder of this section discusses the basics of how to implement .NET and C++ Accelerator applications.

## How to Install Accelerator

You can obtain Accelerator from the Microsoft Connect Web site at [https://connect.microsoft.com/acceleratorv2](http://msdn.microsoft.com/en-us/directx/default.aspx).

To install Accelerator

1. On the Microsoft Connect Web site for Accelerator v2, click the Download link for for Setup.msi and save the file to a convenient location on your hard drive.

2. Run Setup.msi to start the installation Wizard.

3. Follow the wizard instructions.

The installation process is brief and straightforward, and most users can simply accept the default settings.

Accelerator installs the DLLs, libraries, and so on to the following locations:

* For x86 systems, the Accelerator files are installed under Program Files\Microsoft\Accelerator v2.
* For x64 systems, the Accelerator files are installed under Program Files (x86)\Microsoft\Accelerator v2.

Note: To use Accelerator and to follow the steps in this paper, you must have the following installed on your computer:

* Windows 7 or Windows Vista® operating system
* Microsoft Visual Studio 2008 or later
* .NET Framework version 3.5 or later
* DirectX Software Development Kit (SDK)

The DirectX SDK is required only for C++ applications.

## .NET Applications

You can implement .NET-based Accelerator applications by using any .NET language.

To implement a .NET-based Accelerator application

1. Open Microsoft Visual Studio® 2008 (or later version) and create a new .NET project.

The most commonly used project types are Windows® Forms Application, Console Application, and WPF Application.

2. Add a reference to Microsoft.Accelerator.dll.

The DLL is under the following folder:

Program Files\Microsoft\Accelerator v2\bin\Managed

Managed contains Debug and Release folders for the debug and release versions of the DLL.

3. Implement the application in the appropriate source files.

4. Build the application.

5. Copy Accelerator.dll to the folder that contains the executable, typically the project’s bin\debug folder.

Accelerator.dll is under the following folder:

Program Files\Microsoft\Accelerator v2\bin

The bin folder contains separate folders for the x86 and x64 DLLs. The x86 and x64 folders each contain Debug and Release folders, which contain debug and release versions of the DLL.

6. Run the application.

The build configuration depends on which targets you want to use:

* The DirectX 9 target has both x86 and x64 versions, so you can use any build configuration other than Itanium, and run the application on either type of system.
* The multicore target has only an x64 version. You must select either Any CPU or x64 build configuration and run the application on an x64 system.
* For other targets, see the target documentation for details.

The Accelerator operations are all static methods in the **Microsoft.ParallelArrays.ParallelArrays** class. To simplify coding, Accelerator applications typically include a **using** declaration for the **Microsoft.ParallelArrays** namespace and define aliases for the commonly used Accelerator classes:

using Microsoft.ParallelArrays;

using PA = Microsoft.ParallelArrays.ParallelArrays;

using BPA = Microsoft.ParallelArrays.BoolParallelArray;

using DPA = Microsoft.ParallelArrays.DoubleParallelArray;

using FPA = Microsoft.ParallelArrays.FloatParallelArray;

using F4PA = Microsoft.ParallelArrays.Float4ParallelArray;

using IPA = Microsoft.ParallelArrays.IntParallelArray;

## C++ Applications

You can also implement Accelerator applications by using unmanaged C++.

To implement a C++ Accelerator Application

1. Install the latest DirectX SDK, if you have not done so already.

2. Open your application in Visual Studio, or create a new C++ project.

3. Add the following include directives:

#include "Accelerator.h"

#include <D3D9.h>

#include "DX9Target.h"

#include "X64MulticoreTarget.h"

This example includes header files for the DirectX 9 and x64 multicore targets. Add header files for other targets, as required.

4. Open the project’s Properties dialog box.

5. Under **Configuration Properties**, select **C/C++** and, add the following folder to the **Additional Include Directories** field:

Program Files\Microsoft\Accelerator v2\Include

This folder contains Accelerator.h, DX9Target.h and X64MulticoreTarget.h.

6. Under **Configuration Properties**, select **Linker** and, add the folder that contains the appropriate version of Accelerator.lib.

Accelerator.lib is under the following folder:

Program Files\Microsoft\Accelerator v2\lib

The lib folder contains separate folders for the x86 and x64 libraries. The x86 and x64 folders each contain Debug and Release folders, which contain debug and release versions of the library.

7. Under **Linker**, select **Input** and add Accelerator.lib to the **Additional Dependencies** field.

8. Build the application.

9. Copy Accelerator.dll to the folder that contains the executable, typically the project’s bin\debug folder.

10. Run the application.

The build configuration depends on which targets you want to use:

* The DirectX 9 target has both x86 and x64 versions, so you can use the Win32 or x64 build configuration and run the application on either type of system.
* The multicore target has only an x64 version. You must select the x64 build configuration and run the application on an x64 system.
* For other targets, see the target documentation for details.

The Accelerator operations are all functions in the **ParallelArrays** namespace. Accelerator applications typically simplify their code by including the following **using** and **typedef** declarations:

using namespace ParallelArrays; //For Accelerator operations

using namespace MicrosoftTargets; //For the target objects

...

int \_tmain(int argc, \_TCHAR\* argv[])

{

typedef BoolParallelArray BPA;

typedef DoubleParallelArray DPA;

typedef FloatParallelArray FPA;

typedef Float4ParallelArray F4PA;

typedef IntParallelArray IPA;

...

}

# How to Implement Accelerator Applications

The best way to learn how to use Accelerator is by examining some working examples. If you have not done so already, you should read “An Introduction to Accelerator,” which includes a detailed discussion of several very basic examples. This section builds on those examples to provide brief walkthroughs of some more sophisticated Accelerator samples.

* Life implements a cellular automata algorithm often referred to as “Conways Game of Life.”

It demonstrates how to use several mathematical and logical element-wise operations and the PA.Rotate transform.

* Boxcar implements a simple one-dimensional sliding-window filter.

It demonstrates how to use **Shift**, which is one of Accelerator’s key operations.

* 2DConvolution implements a simple image processing algorithm, which uses two-dimensional convolution to blur an image.
* StackAsync is a simple example of asynchronous evaluation for .NET applications, and demonstrates the “fastest target wins” pattern.
* StackAsync\_CPP is a C++ version of StackAsync.
* Boxcar2 is similar to BoxCar, but is implemented by using parameter objects.

For convenience, all of the samples, with the exception of StackAsync\_CPP, are implemented as .NET applications.

**Note:** This paper includes excerpts from the Life, Boxcar, and 2DConvolution samples. The complete samples are available from Accelerator’s Microsoft Connect site.

## Sample Walkthrough: Game of Life

Cellular automata are used for purposes such as cryptography or to simulate natural systems such as certain types of chemical reactions. They are based on a regular grid of cells, each with a defined state. The grid is updated at regular intervals, and the value of each cell in the new grid is based on the states of its nearest neighbors in the original grid. This section is a walkthrough of the Life sample, which is an Accelerator implementation of one of the better-known cellular automata, the Game of Life.

### The Game of Life

The Game of Life is based on a two-dimensional grid. Each cell can have one of two states: alive or dead. An application starts with an initial pattern of live cells and then updates the grid repeatedly so that it evolves through time. Each time the application updates the grid, it determines the state of each cell in the new grid by examining the cell’s nearest neighbors in the current grid and applying the following rules:

* A live cell with two or three live neighbors remains live in the new grid.
* All other live cells become dead in the new grid.
* A dead cell with three live neighbors becomes live in the new grid.
* All other dead cells remain dead in the new grid.

Figure 1 shows how a simple “glider” pattern evolves to the next grid in the sequence. The blue cells are “live,” and the numbers indicate the number of live nearest neighbors for all cells that have them.



Figure 1. Evolution of a Game of Life pattern

### The Life Sample

Life is implemented as a Windows Forms application. Figure 2 shows the Life UI, with a “game” in progress.

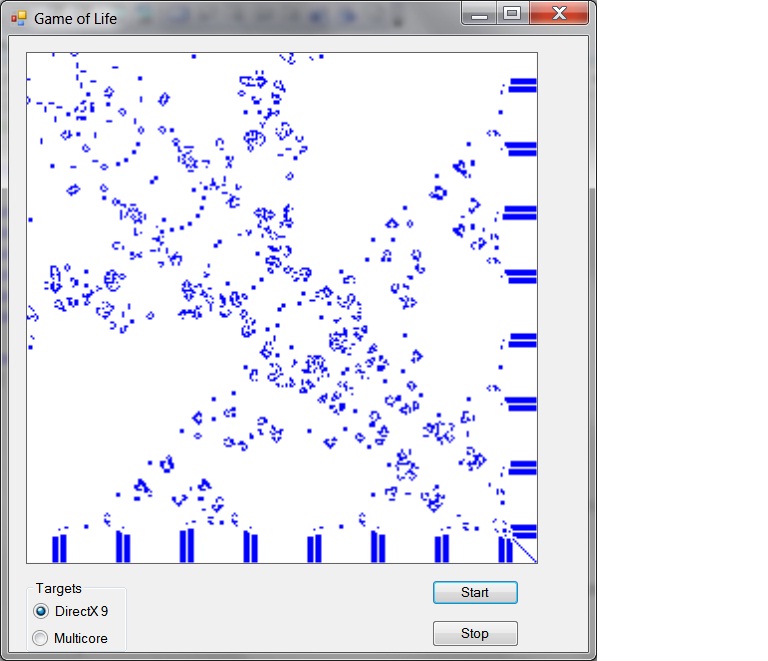


Figure 2. Life user interface

The grid for this version of Life consists of four mirror-image quadrants. To maximize resolution, Life displays only the upper-left quadrant. The UI also includes:

* A **Start** button, which starts a new game.
* A **Stop** button to stop the current game
* A pair of radio buttons, to select which target to use for computation.

When a user clicks **Start**, Life initializes the game using the code in Listing 1.

Listing 1. Life initial grid

for (int i = 0; i < gridSize; i++)

{

currentState[0, i] = 1;

currentState[i, 0] = 1;

currentState[i, i] = 1;

}

imageBox.Image = ArrayToBitmap(currentState);

zeroGrid = new FPA(0.0f, gridDimensions);

oneGrid = new FPA(1.0f, gridDimensions);

twoGrid = new FPA(2.0f, gridDimensions);

threeGrid = new FPA(3.0f, gridDimensions);

To initialize a new game, Life:

1. Generates the initial grid by setting the live cells set to 1.0 and the dead cells to zero.

2. Displays the initial grid in the UI.

*ArrayToBitmap* is a private method that converts the grid values to a blue and white bitmap.

3. Life loads the initial grid into a FloatParallelArray object named *working*Grid, which represents the current grid state in the Accelerator environment.

4. Creates four utility FloatParallelArray objects—zeroGrid, oneGrid, twoGrid, and threeGrid—which have all elements set to 0.0f, 1.0f, 2.0f, or 3.0f, respectively.

Figure 3 shows the upper left part of the initial grid.



Figure 3. Life initial grid

Life also implements three utility methods, And, Or, and Equals, shown in Listing 2.

Listing 2. Life utility methods

FPA And(FPA a, FPA b) { return PA.Min(a, b); }

FPA Or(FPA a, FPA b) { return PA.Max(a, b); }

FPA Equals(FPA a, FPA b)

{

return PA.Cond(PA.CompareEqual(a,b),oneGrid,zeroGrid);

}

The three methods all take two data-parallel array objects, *a* and *b*, as input.

* And uses the element-wise Min operator to determine the minimum value of each pair of elements from *a* and *b*.
* Or uses the element-wise Max operator to determine the maximum value of each pair of elements from *a* and *b*.
* Equals uses the element-wise Cond and CompareEqual operators to determine whether the input elements are equal and convert the result into and array of ones and zeros.

**CompareEqual** returns a **BoolParallelArray** object, whose elements are set to **true** if the corresponding pair of elements from *a* and *b* are equal, and **false** otherwise.

**Cond** returns an object, with each element set to an element of one of the final two arguments. If an element in the first object is **true**, **Cond** returns the corresponding element from the second argument. Otherwise, Cond returns the corresponding element from the third argument. In this case, the initial object is the **BoolParallelArray** returned by **CompareEqual**. Since *oneGrid* and *zeroGrid* contain all 1s and all 0s, respectively:

If a pair of elements from *a* and *b* are equal, **Cond** sets the corresponding element in the final result to 1.

If a pair of elements from *a* and *b* are not equal, **Cond** sets the corresponding element in the final result to 0.

Life creates a System.Windows.Forms.Timer object that fires every 60 ms. Each time the timer fires, Life computes a new grid using the rules described earlier in this section.

Life creates a working grid object by initializing a **FloatParallelArray** object with the current grid. To determine the state of each cell in the new grid, Life uses Rotate to create a FloatParallelArray object, *nearestNeighbors*,that contains the number of nearest neighbors for each cell in the working grid. The code—shown in Listing 3—shifts the grid one cell in each direction and sums the results. Because live cells are set to 1 and dead cells to 0, *nearestNeighbors* represents an array that contains the number of live nearest neighbors for each cell.

Listing 3. Life nearest-neighbors calculation

FPA nearestNeighbors = new FPA(0.0f, gridDimensions);

FPA workingGrid = new FPA(currentState);

for (int gridY = -1; gridY <= 1; gridY++)

{

for (int gridX = -1; gridX <= 1; gridX++)

{

if(!(gridX == 0 && gridY == 0))

{

nearestNeighbors = nearestNeighbors + PA.Rotate(workingGrid,

gridY,

gridX);

}

}

}

Note: Because Rotate wraps the array, the nearest neighbor to the left of a cell on the left edge of the grid is the corresponding cell on the right edge of the grid, and so on.

To create the new grid, Life determines whether cells have the correct number of nearest neighbors to be live in the new grid by using the And, Or, and Equals utility functions to compare sum to the oneGrid, twoGrid, and threeGrid utility objects. The comparison is handled by a single line of code, as shown in Listing 4.

Listing 4. Life

workingGrid = Or(Equals(nearestNeighbors, threeGrid),

And(Equals(nearestNeighbors, twoGrid), workingGrid));

Life then calls **ToArray2D** on the selected target to evaluate the result object. Life can use either **DX9Target** or **X64MulticoreTarget**, with the currently selected target represented by *evalTarget*. **ToArray2D** returns a *currentState* array that contains the updated grid values, which Life then converts to a bitmap and updates the display.

Listing 5. Life evaluation

currentState = evalTarget.ToArray2D(workingGrid);

## Sample Walkthrough: Sliding-Window Filter

Time series—and other similar digital series—are typically represented by a first rank array, and are commonly filtered for purposes such as noise reduction or adding effects. The sample discussed in this section uses a simple sliding-window filter—commonly called a “boxcar” filter—to filter an array.

One way to visualize a boxcar filter is to think of it as a rectangular window that moves through the array one element at a time, as shown in Figure 4.



Figure 4. Visualization of a boxcar filter

For each element in the series, the filtering procedure centers the window on an element and then:

1. Averages all the elements within the window.

2. Assigns that average value to the element that the filter is centered on.

3. Advances the window by one element.

4. Repeats Steps 1–3 until the window reaches the end of the array.

This procedure must be slightly modified at the ends of the array, where the filter extends beyond the first or last array element. Boxcar simply truncates the filter so that it doesn’t extend beyond the array.

More sophisticated versions of this procedure use different filter shapes—for example, a “bell curve” function such as a Gaussian—that weight the elements near the center element more heavily than those near the edge.

The following example shows a conventional iterative implementation of a boxcar filter. For simplicity, it omits the beginning and end of the series, which must be handled somewhat differently.

Listing 6. Conventional filter

for (int i = halfWidth; i < (originalSeries.Length - halfWidth); i++)

{

for (int j = -halfWidth; j <= halfWidth; j++)

{

filteredSeries[i] += originalSeries[i+j]/(2 \* halfWidth + 1);

}

}

Note: Sliding-window filters are usually symmetrical and have an odd number of elements, so that the center of the filter is well defined. The halfWidth value in the example is the number of filter elements on either side of the central element, so the full width of the filter is (2\*halfWidth) + 1.

### The Shift Operation

To implement a sliding-window filter with Accelerator, you must use one of two closely related operations, **Shift** or **ShiftDefault**. Figure 5 shows how **Shift** and **ShiftDefault** work with an array, *a*, of N elements.



Figure 5. The Shift and ShiftDefault operations

The original array is shown in the center of Figure 5. Ignore the shaded elements for the moment. It’s useful to think of the cells in Figure 5 as defining a memory window. If you apply **Shift** or **ShiftDefault** to the original, you shift this window by a specified increment, right or left. Figure 5 shows the result of shifting the window by one or two elements in either direction.

**Important:** When you call **Shift** or **ShiftDefault**, the specified increment applies to the memory window, not the elements. To shift the elements to the right, you must specify a negative increment, and vice versa.

When you apply **Shift** or **ShiftDefault**, the resulting array contains elements that are not defined, as indicated by the shaded elements in Figure 5. The difference between **Shift** and **ShiftDefault** is in how they specify values for these elements.

* **Shift** assigns the nearest element from the original array to the undefined elements.
* **ShiftDefault** assigns a specified constant to the undefined elements.

Figure 6 shows the results of **Shift** and **ShiftDefault** for an increment of +2, with **ShiftDefault** assigning 0 to the undefined elements.



Figure 6. Shift and ShiftDefault, with an increment of two

You can also apply **Shift** and **ShiftDefault** to second rank objects. In that case, you use a two-element **int** array to specify the increment. The array’s first element specifies how far to shift the rows and the second element specifies how far to shift the columns.

### An Obvious (and Not Very Efficient) Sliding-Window Filter Implementation

The most obvious way to implement a sliding window filter with Accelerator is to use **ShiftDefault** to mimic the behavior of the conventional iterative algorithm. Assume that the original series has 1000 elements. Construct a data-parallel filter object that represents the following 1000 element array:

All other elements in the array are set to zero, which creates a filter that is five elements wide, giving it a half-width of two. To filter the original series, use ShiftDefault to shift the filter object one element at a time to the right and assign zeros to the new elements to the left of the boxcar. This effectively moves the “boxcar” from left to right, as shown in Figure 7.



Figure 7. Filter object

Each time the application shifts the filter object, you determine the value of the corresponding element of the filtered series by applying InnerProduct to the filter object and the original series, which calculates the dot product of the two arrays. This approach would result in code that looks something like code in Listing 7.

Listing 7. A functional but inefficient Accelerator filter implementation

for(i=0; i<1000; i++)

{

filterObject = PA.ShiftDefault(filterObject, 0, -i)

filteredSeries += PA.InnerProduct(filterObject, originalSeries);

}

Notes:

* Because ShiftDefault shifts the “memory window” of the object, you must shift the memory window from right to left to shift the filter from left to right.

This is the reason for the -i in the ShiftDefault call.

* This example uses ShiftDefault rather than Shift because, as you shift the position of the filter, you want the new elements to the left of the boxcar to be set to zero.

This code is actually a good example of how a conventional mindset can lead to less than optimal Accelerator code. Consider the following issues:

* The application repeats the two Accelerator operations 1000 times.
* The interesting part of the calculation is limited to the width of the filter itself, which is much shorter than the original array.

Most of the InnerProduct calculation is just a long series of multiply-by-zero operations, which are basically unproductive overhead. However, the application cannot use a shorter filter array, because InnerProduct requires arrays of the same length.

This code will perform the task, but not very efficiently. The following section shows a much better approach.

### Boxcar: An Efficient Sliding-Window Filter Implementation

To use Accelerator effectively, you often need to rethink your approach to array processing. The previous example is conceptually similar to the conventional iterative approach. However, it requires a large array to represent a much smaller filter, which leads to a relatively inefficient calculation.

A much better approach—which is used by the Boxcar sample—is to create an n-element kernel array to describe the filter, where n is the width of the filter and is normally an odd number. For a boxcar filter, the elements of the kernel array are all set to a constant value, usually 1/(filter width) so that the filtering operation doesn’t change the amplitude of the series. You can also create kernel arrays for other filter shapes, such as the Gaussian kernel discussed in “Two-Dimensional Convolution” later in this document.

Figure 8 shows the Boxcar UI.

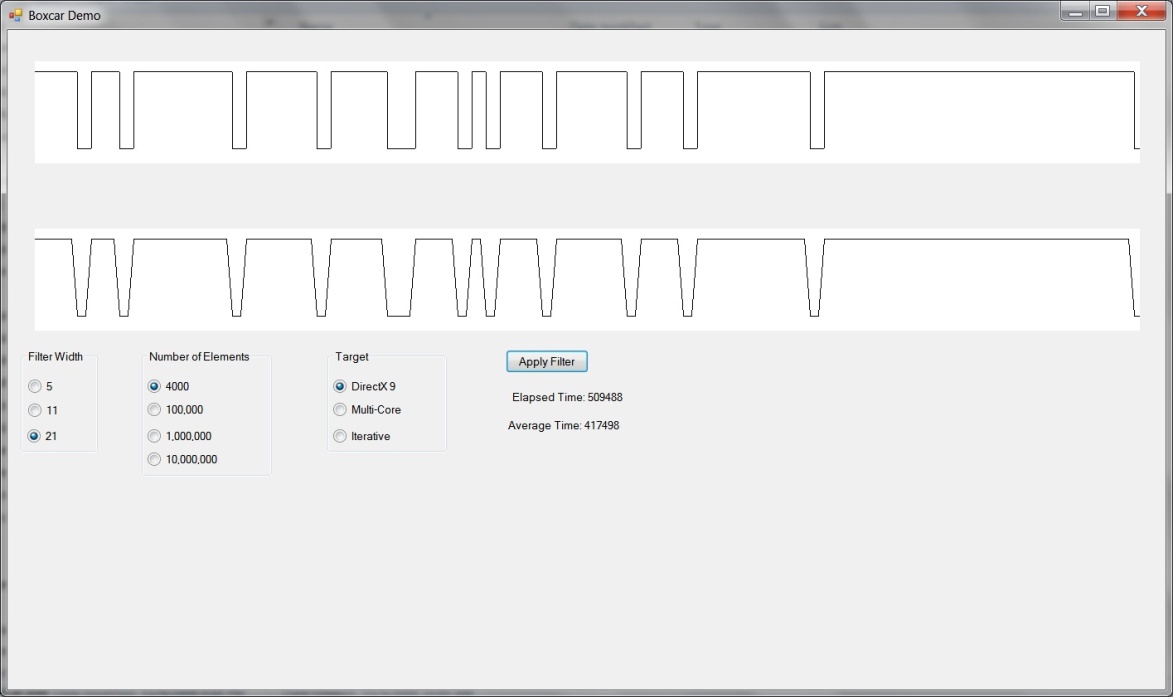


Figure 8. Boxcar UI

Boxcar generates a random one-dimensional square wave, shown in the upper image, and then applies the filter, shown in the lower image. The elapsed time for the operation is shown below the **Apply Filter** button. The radio buttons allow you to select several filter widths and array lengths. The sample also includes a conventional iterative calculation for comparison.

The core of the application is the filtering operation. The following procedure describes how Accelerator performs the operation.

To perform the filtering operation

1. Use ShiftDefault to shift the original array n/2 to the left.

2. Use the \* operator—which is equivalent to Multiply—to multiply the shifted array by the first element of the kernel array.

3. Shift the original array (n/2 - 1) to the left.

4. Multiply the shifted array by the second element of the kernel array, and then add the resulting object to the object from Step 2.

5. Continue this process until you reach the end of the kernel array.

6. Pass the result object to **ToArray**, which returns the filtered array.

Listing 8 shows Boxcar’s Accelerator filtering implementation. It applies the filter (*kernel*) to the original array, which is represented by a **FloatParallelArray** object, *fpTimeSeries.*

Listing 8. An efficient Accelerator filter implementation

for (int i = 0; i < kernel.Length; i++)

{

fpResult += PA.Shift(fpTimeSeries, i) \* kernel[i];

}

filteredSeries = mcTarget.ToArray1D(fpResult);

Mathematically, this code is equivalent to the example in the previous section. However, it iterates over the width of the filter—not the entire length of the original series—and there are no unnecessary multiply-by-zero operations. This results in a much more efficient computation.

**Note:** Boxcar does not use the DirectX 9 target because the length of a 1-dimensional array is limited to the maximum texture width, which is typically only 4K or 8K, depending on the video adaptor. This is too small for the processing efficiency of the GPU to overcome the cost of transferring the code and data across the PCIe bus. The multicore target handles arrays of any length.

The example above provides several advantages over conventional iterative calculations:

* Accelerator can provide much better performance than conventional sequential calculations, as long as you are processing a sufficiently large array.

Boxcar includes a conventional iterative computation—not discussed here—for comparison.

* The code is simpler.

The core of the calculation is a single line of code.

* Shift and ShiftDefault automatically extend the array, so there is usually no need for special-case code to handle the array ends.

## Sample Walkthrough: Two-Dimensional Convolution

The 2DConvolution sample uses two-dimensional convolution to smooth a simple bitmap. The approach is similar to that used in the boxcar sample, but 2DConvolution applies a Gaussian kernel to a two-dimensional array.

The 2DConvolution application:

1. Creates a simple bitmap, consisting of 300 randomly placed solid black ellipses of various sizes.

2. Convolves the bitmap with a Gaussian function, creating a blurred “halo” around each ellipse.

A Gaussian function defines a standard bell curve, as follows:

where σ specifies the curve’s width and a specifies the location of the curve’s peak.

The sample creates the bitmap by using the System.Drawing.Graphics.FillEllipse method and a random number generator to create 300 ellipses of varying position and size. See the sample’s CreateTestBitmap method for details.

Figure 9 shows the 2DConvolution UI.

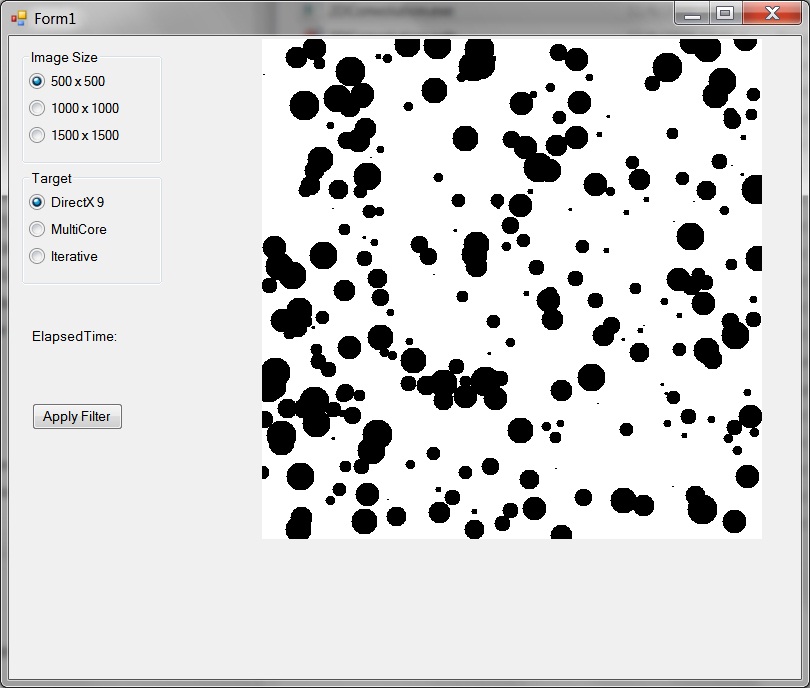


Figure 9. 2DConvolution UI

The UI allows you to select an image size and the target to be used to perform the operation. It includes a standard iterative calculation for comparison.

When the user clicks **Apply Filter**, 2DConvolution calculates a Gaussian kernel array, *filter*, as shown in Listing 9.

Listing 9. 2DConvolution kernel calculation

for (i = 0; i < filterWidth; i++)

{

filter[i] = (float)Math.Exp(-(i - halfWidth) \* (i - halfWidth) / (2 \* sigma \* sigma));

sum += filter[i];

}

The halfWidth parameter determines the kernel length, and *sigma* determines the filter’s shape.

To blur the bitmap, 2DConvolution uses an algorithm similar to the one used by Boxcar. The primary differences are that 2DConvolution uses a Gaussian kernel and applies it to a two-dimensional array.

Listing 10 shows the core of 2DConvolution’s, the convolution algorithm.

Listing 10. 2DConvolution filtering operation

FPA data = new FPA(imageArray);

// Convolve in X direction.

FPA smoothX = new FPA(0, data.Shape);

for (i = 0; i < filterWidth; i++)

{

smoothX += A.Shift(data, 0, i - halfWidth) \* filter[i];

}

// Convolve in Y direction.

FPA result = new FPA(0, data.Shape);

for (i = 0; i < filterWidth; i++)

{

result += A.Shift(smoothX, i - halfWidth, 0) \* filter[i];

}

filteredArray = evalTarget.ToArray2D(result);

bmFiltered = ArrayToBitmap(filteredArray);

imageBox.Image = bmFiltered;

The convolution operation:

* Uses **Shift** to apply the filter to the image array, *data*.

The complete operation takes place in two stages, one pass for the x direction and one pass for the y direction. One of the useful properties of a Gaussian kernel is that it can be applied independently to each dimension of the image, so the x and y operations can be performed in either order.

* Passes the result object to the selected target object’s **ToArray** method, which returns a filtered array.
* Converts the array to a bitmap and updates the image.

This target can use the DirectX 9 as well as the multicore target. The DirectX 9 target can handle 2-dimensional arrays as large as 4K x 4K or 8K x 8K, depending on the video adaptor. This is large enough for the GPU’s processing efficiency to overcome the cost of transferring data and code across the PCIe bus. The application also includes a conventional iterative calculation for comparison.

Figure 10 shows a part of an original image and the corresponding portion of the blurred image.



Figure 10. Original and filtered images

## Sample Walkthrough: Asynchronous Evaluation with the Managed Target API

The Stack\_Async sample is similar to the StackArrays discussed in “An Introduction to Accelerator v2.” It stacks a pair of 4000 x 4000 element 2-D arrays by using the same graph construction code as StackMany. However, Stack\_Async evaluates the result asynchronously on two targets, using the “fastest target wins.”

To implement “fastest target wins”

1. Start the evaluation asynchronously on all available targets.

2. Wait for the fastest target to finish.

3. Cancel the evaluations running on the remaining targets.

The following example is an excerpt of the complete sample, which can be found in Appendix B.

...

FPA fpInput1 = new FPA(inputArray1);

FPA fpInput2 = new FPA(inputArray2);

FPA fpStacked = PA.Add(fpInput1, fpInput2);

FPA fpResult = PA.Divide(fpStacked, 2.0f);

evalTime.Start();

dxAsyncResult = dxTarget.BeginToArray(fpResult, out dxStackedArray);

mcAsyncResult = mcTarget.BeginToArray(fpResult, out mcStackedArray);

int firstFinished = WaitHandle.WaitAny(new WaitHandle[]

{dxAsyncResult.AsyncWaitHandle , mcAsyncResult.AsyncWaitHandle});

if (firstFinished == 0) //dx returned first

{

mcAsyncResult.Cancel();

mcCancelled = true;

mcAsyncResult.AsyncWaitHandle.WaitOne();

stackedArray = dxStackedArray;

}

else //mc returned first

{

dxAsyncResult.Cancel();

dxCancelled = true;

dxAsyncResult.AsyncWaitHandle.WaitOne();

stackedArray = mcStackedArray;

}

dxTarget.Dispose();

mcTarget.Dispose();

...

Stack\_Async then calls each target’s **BeginToArray** method to start the evaluations, and passes in:

* The result object that is to be evaluated.

Targets work with a copy of the object to be evaluated, so you can safely pass *fpResult* to multiple targets.

* An output array to contain the processed data.

Each target should have a separate output buffer, or you risk creating a race condition. A target can write to its output buffer at any time, and there is no way to synchronize access.

The **BeginToArray** calls immediately return **IAcceleratorAsyncResult** interfaces. The interfaces provide **AsyncWaitHandle** objects that the targets signal to notify the application that evaluation is complete. Stack\_Async passes the handles to **WaitHandle.WaitAny**, and waits for the fastest target to complete its evaluation. You can also receive completion notification through a callback delegate. For an example of this approach, see “Sample Walkthrough: A Sliding-Window Filter Implemented with Parameter Objects” later in this paper.

After **WaitAny** returns, Stack\_Async calls **IAcceleratorAsyncResult.Cancel** to cancel the remaining evaluation. The target signals a completion event when cancellation completes, just as it does when evaluation completes. Stack\_Async uses **IAcceleratorAsyncResult.AsyncWaitHandle.WaitOne** to wait on that completion event, which ensures that everything is properly cleaned up before proceeding.

After cancellation is complete, Stack\_Async calls **Dispose** to destroy the target objects. Targets require significant resources, so you should destroy them as soon as you are finished. The application then prints some of the processed data, which is not shown in the example.

## Sample Walkthrough: Asynchronous Evaluation with the C++ Target API

The differences in the languages require the .NET and C++ APIs to handle asynchronous evaluation somewhat differently. This section is a walkthrough of a C++ application named Stack\_Async\_CPP, which is a simplified version of Stack\_Async, that evaluates on only one target. It includes excerpts of the complete sample, which can be found in Appendix B.

The following example shows the key code from Stack\_Async\_CPP.

HANDLE evalCompleteEvent = CreateEvent(NULL, false, false, NULL);

DWORD nStart = GetTickCount();

evalTarget->ToArray(fpResult,

\*stackedArray,

arrayHeight,

arrayWidth,

arrayWidth\*sizeof(float),

evalCompleteEvent);

DWORD nReturn = GetTickCount();

DWORD dwRet = WaitForSingleObject(evalCompleteEvent, INFINITE);

DWORD nComplete = GetTickCount();

evalTarget->Delete();

printf("Return time: %d Completion Time: %d\n",(nReturn - nStart),

(nComplete - nStart));

for(int i = 0; i< 10;i++)

{

printf("[%d, 0] : %f\n",i, stackedArray[i][0]);

}

To evaluate fpResult, Stack\_Async\_CPP:

1. Creates an event handle, *evalCompleteEvent*.

2. Starts the evaluation by calling **ToArray**, and passing it:

The result object that is to be evaluated.

A pointer to the array that will receive the processed data.

The array’s height, width, and pitch.

The event handle.

**ToArray** immediately returns.

3. Calls **WaitForSingleObject** to wait on the completion event.

For simplicity, Stack\_Async\_CPP simply blocks until evaluation is complete. Like the .NET version of the application, the result array does not yet contain valid data. When the target completes the evaluation, it populates the result array with the processed data and signals the event, which causes **WaitForSingleObject** to return.

4. Calls **Target->Delete** to free the target object.

Target objects require substantial resources, so you should delete them as soon as you are finished.

5. Prints the timing data and the first ten elements of the processed array.

## Sample Walkthrough: A Sliding-Window Filter Implemented with Parameter Objects

The BoxCar sample discussed earlier applies a simple sliding window filter to a single input array. If you need to apply the same processing to multiple input arrays, you could simply repeat the operations for each array. However, this approach requires Accelerator to rebuild the expression graph for each input array, even though the graph itself does not change. It just has a different data-parallel array object attached to the input data node.

Parameter objects serve as placeholders in an expression graph. You use them in place of data-parallel objects to construct the expression graph. Before evaluation, you bind a data-parallel array object to each parameter object. To run the evaluation again with different data, just bind a different set of data-parallel array objects. Accelerator just attaches them to the appropriate data nodes; it does not need to rebuild the graph.

BoxCar2 is a console application that takes ten input time series, and applies a boxcar filter to each series. The processing code is the same as that used by Boxcar. Because multiple evaluations can take a substantial amount of time, BoxCar2 uses asynchronous evaluation, to allow the application to perform other work until all the arrays have been processed.

This section is a walkthrough of Boxcar2. It includes excerpts of the complete sample, which can be found in Appendix C.

BoxCar2 creates ten input arrays that contain noisy sine waves, as follows:

for (i = 0; i < numArrays; i++)

{

inputArrays[i] = new float[arrayLength];

for (j = 0; j < arrayLength; j++)

{

inputArrays[i][j] = (float)(Math.Sin((double)j / 10.0) +

ranf.NextDouble() / 5.0);

}

fpInputs[i] = new FPA(inputArrays[i]);

}

Each input array is loaded into a **FloatParallelArray** object.

The graph-construction code is similar to that used by BoxCar, with one key exception:

FPAP fpInputParam = new FPAP("input array");

...

for (i = 0; i < kernel.Length; i++)

{

fpResult += PA.Shift(fpInputParam, i) \* kernel[i];

}

Boxcar2 passes a **FloatParallelArrayParam** object, *fpInputParam*, to **Shift**. **FloatParallelArrayParam** inherits from **FloatParallelArray**, and is basically a **FloatParallelArray** object that initially has no associated data array. The data is assigned later.

To evaluate the result for a particular input array

1. Call **FloatParallelArrayParam.Bind** to bind the **FloatParallelArray** object that represents the input array to *fpInputParam*

2. Call **ToArray**.

The simplest approach is to iterate through the set of input arrays and wait for each **ToArray** call to return. However, this approach could block your main thread from doing any other work for a considerable amount of time. Instead, BoxCar2 uses asynchronous evaluation, which allows the application to do something else while the arrays are processed.

AsyncCallback callback = new AsyncCallback(MCCallback);

...

for (i = 0; i < numArrays; i++)

{

fpInputParam.Bind(fpInputs[i]);

mcTarget.BeginToArray(fpResult,

out outputArrays[i],

mcTarget,

callback);

}

BoxCar2 uses a callback delegate to receive completion notifications, so it passes **BeginToArray**:

* The result object that is to be evaluated.
* An output array to contain the processed data.
* The target object.
* A callback delegate.

**BeginToArray** returns immediately, so this code starts all the evaluations at nearly the same time, and then returns control to the main thread while the evaluations proceed on worker threads. When each evaluation completes, the target calls *MCCallback*.

class Boxcar2

{

static int evalCount;

static object completionLock;

...

static void MCCallback(IAsyncResult result)

{

lock (completionLock)

{

evalCount--;

}

}

}

BoxCar2 uses a counter, *evalCount*, to keep track of how many input arrays have been processed. Initially, *evalCount* is set to the number of input arrays, and each time *MCCallback* is called, it decrements the counter. The lock avoids a possible race condition with the primary thread, which reads the counter to determine when all evaluations are complete.

While the evaluations are underway BoxCar2 can do other work.

while (waiting)

{

Thread.Sleep(50);

lock (completionLock)

{

waiting = (evalCount > 0);

}

}

BoxCar2 uses polling to determine when all the arrays have been processed. The application perform its work in a **while** loop—for simplicity, it simulates work by sleeping for 50ms—and periodically checks *evalCount*. When *evalCount* is 0, evaluation is complete and the main thread can continue. The final part of the application prints the results, and is not shown here.

# Resources

This section provides links to additional information about Accelerator.

#### Accelerator Resources

Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses

<http://research.microsoft.com/research/pubs/view.aspx?tr_id=1040>

Microsoft Accelerator Documentation

An Introduction to Microsoft Accelerator  
Microsoft Accelerator v2 Programming Guide  
Microsoft Accelerator Target Implementers’ Guide  
[http://research.microsoft.com/Accelerator/](https://connect.microsoft.com/acceleratorv2)

Microsoft Accelerator Updates and Software Availability News

[http://connect.microsoft.com/acceleratorv2](http://research.microsoft.com/research/downloads/Details/25e1bea3-142e-4694-bde5-f0d44f9d8709/Details.aspx)

Microsoft Research Accelerator Project Download

[http://research.microsoft.com/research/downloads/Details/25e1bea3-142e-4694-bde5-f0d44f9d8709/Details.aspx](http://research.microsoft.com/en-us/collaboration/tools/dryad.aspx)

#### Related Resources

Calling Synchronous Methods Asynchronously

[http://msdn.microsoft.com/en-us/library/2e08f6yc.aspx](http://en.wikipedia.org/wiki/Embarrassingly_parallel)

Conway's Game of Life

<http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life>

Directed acyclic graph

[http://en.wikipedia.org/wiki/Directed\_acyclic\_graph](http://research.microsoft.com/Accelerator/)

DirectX Developer Center

[http://msdn.microsoft.com/en-us/directx/default.aspx](http://connect.microsoft.com/acceleratorv2)

Dryad and DryadLINQ for Data Intensive Research

[http://research.microsoft.com/en-us/collaboration/tools/dryad.aspx](http://msdn.microsoft.com/en-us/library/2e08f6yc.aspx)

Embarrassingly Parallel

[http://en.wikipedia.org/wiki/Embarrassingly\_parallel](http://en.wikipedia.org/wiki/Directed_acyclic_graph)

# Appendix A: Selected Method Descriptions

This section contains detailed descriptions of how to use the more complicated Accelerator operations: transforms and linear algebraic operations. The associated methods are in alphabetic order. For more details, see the Accelerator help file.

|  |  |  |  |
| --- | --- | --- | --- |
| [AddDimension](#_AddDimension) | [InnerProduct](#_InnerProduct) | [Rotate](#_Rotate) | [Transpose](#_Transpose) |
| [DropDimension](#_DropDimension)  Evaluate | [OuterProduct](#_OuterProduct) | [Section](#_Section) |  |
| [Expand](#_Expand) | [Pad](#_Pad) | [Shift/ShiftDefault](#_Shift_and_ShiftDefault) |  |
| [Gather](#_Gather) | [Replicate](#_Replicate) | [Stretch](#_Stretch) |  |

## AddDimension

AddDimension(a, dimension) transforms a first rank data-parallel array object into a second rank object by adding an empty dimension. The shape of the second rank object is controlled by the method’s dimension parameter:

* If dimension is set to 0, AddDimension creates a second rank object with one row, which contains the elements of the starting array.
* If dimension is set to 1, AddDimension creates a second rank object with one column, which contains the elements of the starting array.

For example, if a represents the following first rank array:

AddDimension(a,0) yields an object that represents the following 1x3 second rank array:

AddDimension(a,1) yields an object that represents the following 3x1 second rank array:

## DropDimension

DropDimension(a, dimension) transforms a second rank data-parallel array object into a first rank object by removing a dimension. The dimension to be removed must be empty.

The dimension to be removed is controlled by the method’s dimension parameter, as follows:

* If dimension is set to 0, DropDimension transforms a second rank object with one row into a first rank object, which contains the elements of the row.
* If dimension is set to 1, DropDimension transforms a second rank object with one column into a first rank object, which contains the elements of the column.

## Evaluate

**Evaluate**(*a*) returns a data-parallel array object that represents the same expression as *a*. However, this new data-parallel array alters how all targets will compute the value of an expression containing it. When a result of a call to **Evaluate** is encountered by a target, if the target has a cached value for that expression, that value will be used. If there is no value, an intermediate result will be computed and stored for future use. Because values are stored, be sure to dispose the result of a call to **Evaluate** so that resources can be returned to the target processor.

## Expand

Expand(a, before, after) increases the dimensions of a data-parallel array object. The before and after parameters are integer arrays. For first-rank data-parallel objects, before and after both have a single element, which specifies how many elements are to be added to the beginning or end of a. For second rank objects, before and after have two elements, as follows:

* The first element of the before array specifies how many rows to add above the first row of the original object.
* The second element of the before array specifies how many columns to add before the first column of the original array.
* The first element of the after array specifies how many rows to add below the last row of the original object.
* The second element of the after array specifies how many columns to add after the last column of the original array.

Expand assigns values to the new elements by wrapping the elements of the original object.

The simplest way to understand how Expand works is to look at several specific cases.

#### First Rank Objects

Assume that a represents the following first-rank array:

Set before to (1) and after to (0). Expand(a, before, after) yields the following:

Setting before to (0) and after to (2) yields the following:

#### Second Rank Objects

For second rank objects, assume that a represents the following array:

Set before to (1,0) and after to (0,0). Expand(a, before, after) yields the following 3x2 array:

Setting before to (0,1) and after to (0,0) yields the following 2x3 array:

Setting before to (0,0) and after to (1,0) yields the following 3x2 array:

Setting before to (1,1) and after to (0,0) yields the following 3x3 array:

Setting before to (2,0) and after to (0,0) yields the following 4x2 array:

## Gather

Gather (a, positions) creates a new object that contains a specified subset of the elements of the original object. The new object has the same rank as the original, but fewer elements.

The elements to be included in the new object and their location in the object are specified by one or two IntParallelArray positions objects, which contain the indices of the elements to be included in the new array. The positions objects also determine the shape of the new object and the location of the selected elements. The rank of the original object determines the number of positions objects that are required.

#### First Rank Objects

For a first rank object of dimension N, the positions object represents an array of one to N integers. The value of each integer must between 0 and (N-1), and values can be repeated. The new object contains the elements of the original whose indices are in the positions object, in the order that they are listed in that object.

For example, assume that a represents the following first rank array:

Set positions to the following:

Gather(a, positions) yields the following:

#### Second Rank Objects

A second rank object of shape N x M requires two second rank positions objects, the first for row indices and the second for column indices.

The positions objects are a matched pair, which specify:

* The elements that are to be in the new object.

The row positions object contains the row indices of the selected elements, and the column positions object contains the corresponding column indices.

* The position of each element in the new object.

The position is determined by the location of the indices in the row and column positions objects. For example if the (1, 0) elements of the row and column positions objects contain 2 and 3, respectively, element (2,3) of the original object becomes element (1, 0) of the new object.

* The shape of the new object, which is identical to the shape of the row and column positions objects.

The positions objects must satisfy the following constraints:

* The positions objects must be the same shape.
* The shape cannot exceed the bounds of the original object.

Gather cannot, for example produce a 4x2 object from a 2x2 object.

* Each element of the row positions object must be set to integer between 0 and (N-1).
* Each element of the column positions object must be set to integer between 0 and (M-1).

For example, assume that the original object represents the following second rank array:

To extract the 2 x 2 matrix from the center of a, specify row and column positions objects that represent the following arrays:

Gather(a, rowIndex, colIndex) yields the following:

## InnerProduct

InnerProduct(a1, a2) performs matrix and vector multiplication on **DoubleParallelArray**, FloatParallelArray and IntParallelArray objects. The method takes two data-parallel array objects and returns a new object of the same type, as follows:

* If both input objects are second rank, InnerProduct performs standard matrix multiplication on the objects and returns a new second-rank object of the appropriate type and shape.
* If one object is first rank and the other is second rank, InnerProduct performs standard vector-matrix multiplication on the object and returns a new first rank object of the appropriate dimensions.
* If both objects are first rank, InnerProduct returns a scalar object containing the scalar product of the two arrays.

#### Matrix-Matrix Multiplication

Assume that a1 and a2 represent the following two IntParallelArray objects:

InnerProduct(a1, a2) yields the following:

#### Matrix-Vector Multiplication

Assume that a1 and a2 represent the following two arrays:

InnerProduct(a1, a2) yields the following:

#### Scalar Multiplication

Assume that a1 and a2 represent the following two arrays:

InnerProduct(a1, a2) yields the following:

## OuterProduct

OuterProduct(a, b) returns the outer product of two first-rank objects. The outer product is sometimes called a tensor or dyadic product, and it is defined as follows:

The number of columns in the resulting object is determined by the length of the first object, and the number of rows is determined by the length of the second object.

Assume that a and b are data parallel array objects that represent the following arrays:

OuterProduct(a,b) yields the following:

## Pad

Pad(a, before, after, padValue) increases the dimensions of a data parallel array object and assigns a specified value to the new elements.

Similar to Expand, Pad takes before and after integer arrays that specify where the extra elements are to be placed. Pad’s fourth parameter is padValue, which specifies the value to be assigned to the extra elements.

#### First Rank Objects

Assume that a represents the following first-rank array:

Set before to (1), after to (0), and padValue to 7. Pad(a, before, after, padValue) yields the following array:

Setting before to (0) and after to (2) yields the following:

#### Second Rank Objects

For second rank objects, assume that a represents the following array:

Set before to (1,0), after to (0,0), and padValue to 42. Pad(a, before, after, padValue) yields the following 3x2 array:

## Replicate

Replicate(a, dimensions) increases the dimensions of an array and assigns values to the new elements by placing the original array in the upper left corner and using it to tile the remainder of the array. The dimensions parameter is an integer array that specifies the dimensions of the new array.

For second rank objects, assume that a represents the following array:

If you set dimensions to (3, 5), Replicate(a, dimensions) yields the following array:

For more flexibility in expanding arrays, use the Expand or Pad transforms.

## Rotate

Rotate(a, counts) cyclically permutes the rows or columns of an array without changing the shape or creating new elements.

The counts parameter is an integer array that specifies the number of elements by which the array is to be rotated. It has one element when used for first rank data-parallel objects and two elements when used for second rank objects.

#### First Rank Objects

Assume that a represents the following first-rank array:

Set count to (2). Rotate (a, count) cyclically permutes the elements by two, yielding the following array:

#### Second Rank Objects

For second rank objects, assume that a represents the following second rank array:

Set count to (1, 0). Rotate (a, count)cyclically permutes the rows by one, which yields the following array:

Setting count to (0, 2) cyclically permutes the columns by two, which yields the following array:

## Section

**Section**(*a, specifier, [specifier2]*) creates a data-parallel array object that contains a specified subset of the elements of the original object, as follows:

* The transformed array consists of regularly spaced sections of the original array.
* The organization of the sections is based on their positions in the original array.
* The transformed array is usually smaller than the original, but can be larger.
* **Section** is typically faster than **Gather**.

Section allows you to create a rich set of subarrays, but the requirement that the selected elements be regularly spaced limits the transform’s flexibility. For more flexibility in selecting subsets of an array, use Gather.

The second parameter is a **SectionSpecifier** object that specifies how the slices are to be selected. Second rank objects require two **SectionSpecifier** objects. The first applies to the input object’s rows and the second applies to the columns.

* Start specifies the array index of the first slice.
* Count specifies how many slices to take.

If Count causes Section to go beyond the extent of the original array, Section sets the remaining elements to the value of the last element in the original array—sometimes referred to as a “clamped” set. If Count is set to -1, Section takes as many slices as possible, within the bounds of the original array.

* Stride specifies spacing of the slices.

For example, if Stride is set to three, Section selects every third element.

#### First Rank Objects

For first rank objects, Section takes a single **SectionSpecifier** object. Assume that a represents the following first-rank array:

For the following **SectionSpecifier** settings: Start = 0, Count = 2, Stride = 3. Section(a, specifier) yields the following:

Set the members of specifier to: Start = 0, Count = 5 and Stride = 3. Section(a, specifier) yields the following:

Because Count forced Section to go beyond the bounds of the original array, clamping sets the final two elements to the last value in the array.

#### Second Rank Objects

For second rank objects, Section takes two **SectionSpecifier** objects. The first specifies the selected rows, and the second specifies the selected columns. To be selected, an element must be in both a column slice and a row slice. The dimensions of the transformed array are thus (row count, column count). Assume that a represents the following 4x9 array:

Set the **SectionSpecifier** objects as follows:

* rowSlices: Start = 1, Count = 2, Stride = 2
* columnSlices: Start = 0, Count = 4, Stride = 3

Section(a, rowSlices, columnSlices) yields the following:

Set the **SectionSpecifier** objects as follows:

* rowSlices: Start = 1, Count = 3, Stride = 2
* columnSlices: Start = 0, Count = 3, Stride = 2

Section(a, rowSlices, columnSlices) yields the following:

## Shift and ShiftDefault

Shift(a, counts) and ShiftDefault(a, defaultValue, counts) move the memory window represented by a data-parallel array object in a specified direction. The shape and dimension of the underlying array remain the same, but the elements are shifted left, right, up, or down relative to the original array. These two transforms are particularly useful for tasks such as filtering digital streams or images.

Moving the memory window transforms some elements out of the array and creates a corresponding set of unspecified elements on the opposite side of the array. The difference between Shift and ShiftDefault is how they assign values to these unspecified elements:

* Shift assigns the values of the closest elements from the original array to the unspecified elements.
* ShiftDefault assigns a specified constant value to the unspecified elements.

Shift and ShiftDefault have a counts parameter, which specifies how far and in which direction to shift the object’s memory window. The counts parameter is a one- or two-element integer array, depending on the object’s rank. A positive counts value shifts the window to the right or down, which has the net effect of shifting the elements to the left or up. The methods subtract the counts values from the indices of each element and discard any elements with negative indices.

ShiftDefault takes an additional defaultValue parameter that specifies the value to be assigned to the new elements that are created by the transform.

#### First Rank Objects

For first rank objects, counts is a one-element array. Assume that a represents the following first-rank array:

Set counts to (2). Shift(a, counts) yields the following:

Set counts to (2) and defaultValue to 0. ShiftDefault(a, defaultValue, counts) yields the following:

Setting counts to -2 yields the following:

#### Second Rank Objects

For second rank objects, counts is a two-element array. The first element specifies how to transform the rows, and the second element specifies how to transform the columns. Assume that a represents the following second rank array:

Set counts to (1,-1) and defaultValue to 0. ShiftDefault(a, defaultValue, counts) yields the following:

## Stretch

Stretch(a, multiplicity) expands an object by replicating the elements of the original array a specified number of times. The multiplicity parameter is a one- or two-element integer array, depending on the rank of a. The multiplicity parameter specifies how many instances of each element are to be in the new object.

#### First Rank Objects

Assume that a represents the following array:

Set multiplicity to 3. Stretch(a, multiplicity) yields the following:

#### Second Rank Objects

Assume that a represents the following array:

Set multiplicity to (3, 2). Stretch(a, multiplicity) yields the following:

## Transpose

Transpose(a, dimensions) performs standard matrix transposition. If a is the original object, aT[i, j] = a[j, i]. Transpose is implemented only for second rank data-parallel objects.

Assume that a represents the following array:

Transpose(a) yields the following:

# Appendix B: Complete Code for Stack\_Async and Stack\_Async\_CPP

This appendix contains the complete source code for the Stack\_Async and Stack\_Async\_CPP applications. Both are console applications, so you can run the samples by creating a console application of the appropriate type and pasting the source code. For more details on how to set up .NET and C++ Accelerator applications, see “Accelerator Programming Fundamentals” earlier in this paper.

## Stack\_Async Sample Code

using System;

using System.Diagnostics;

using System.Threading;

using Microsoft.ParallelArrays;

using FPA = Microsoft.ParallelArrays.FloatParallelArray;

using PA = Microsoft.ParallelArrays.ParallelArrays;

namespace Stack\_Async

{

class StackAsync

{

static void Main(string[] args)

{

int arrayWidth = 4000;

int arrayHeight = 4000;

Random ranf = new Random();

Stopwatch evalTime = new Stopwatch();

DX9Target dxTarget = new DX9Target();

X64MulticoreTarget mcTarget = new X64MulticoreTarget();

IAcceleratorAsyncResult dxAsyncResult, mcAsyncResult;

bool dxCancelled = false;

bool mcCancelled = false;

float[,] inputArray1 = new float[arrayHeight, arrayWidth];

float[,] inputArray2 = new float[arrayHeight, arrayWidth];

float[,] stackedArray = new float[arrayHeight, arrayWidth];

float[,] dxStackedArray = new float[arrayHeight, arrayWidth];

float[,] mcStackedArray = new float[arrayHeight, arrayWidth];

for(int i=0;i<arrayHeight;i++)

{

for(int j=0;j<arrayWidth;j++)

{

inputArray1[i,j] = (float) ranf.NextDouble();

inputArray2[i,j] = (float) ranf.NextDouble();

}

}

FPA fpInput1 = new FPA(inputArray1);

FPA fpInput2 = new FPA(inputArray2);

FPA fpStacked = PA.Add(fpInput1, fpInput2);

FPA fpResult = PA.Divide(fpStacked, 2.0f);

evalTime.Start();

dxAsyncResult = dxTarget.BeginToArray(fpResult, out dxStackedArray);

mcAsyncResult = mcTarget.BeginToArray(fpResult, out mcStackedArray);

Console.WriteLine("Evaluation starts at: {0} ms", evalTime.ElapsedMilliseconds);

int firstFinished = WaitHandle.WaitAny(new WaitHandle[] {dxAsyncResult.AsyncWaitHandle , mcAsyncResult.AsyncWaitHandle});

if (firstFinished == 0) //dx returned first

{

mcAsyncResult.Cancel();

mcCancelled = true;

mcAsyncResult.AsyncWaitHandle.WaitOne();

stackedArray = dxStackedArray;

}

else //mc returned first

{

dxAsyncResult.Cancel();

dxCancelled = true;

dxAsyncResult.AsyncWaitHandle.WaitOne();

stackedArray = mcStackedArray;

}

dxTarget.Dispose();

mcTarget.Dispose();

Console.WriteLine("Evaluation finished at: {0} ms", evalTime.ElapsedMilliseconds);

Console.WriteLine("DX cancelled: {0} MC cancelled: {1}\n", dxCancelled, mcCancelled);

for (int i = 0; i < 10; i++)

{

Console.WriteLine("stackedArray[{0},0]: {1}", i, stackedArray[i, 0]);

}

Console.WriteLine("...");

Console.ReadKey();

}

}

}

## Stack\_Async\_CPP Sample Code

#include "stdafx.h"

#include "Accelerator.h"

#include <D3D9.h>

#include "DX9Target.h"

#include "X64MulticoreTarget.h"

#include <math.h>

using namespace ParallelArrays;

using namespace MicrosoftTargets;

int \_tmain(int argc, \_TCHAR\* argv[])

{

typedef FloatParallelArray FPA;

const int arrayWidth = 4000;

const int arrayHeight = 4000;

float (\*inputArray1)[arrayWidth] = new float[arrayHeight][arrayWidth];

float (\*inputArray2)[arrayWidth] = new float[arrayHeight][arrayWidth];

float (\*stackedArray)[arrayWidth] = new float[arrayHeight][arrayWidth];

Target& evalTarget = CreateDX9Target();

for(int i = 0;i<arrayHeight;i++)

{

for(int j=0;j<arrayWidth;j++)

{

inputArray1[i][j] = (float) rand();

inputArray2[i][j] = (float) rand();

}

}

FPA fpInput1 = FPA((float\*)inputArray1, arrayHeight, arrayWidth);

FPA fpInput2 = FPA((float\*)inputArray2, arrayHeight, arrayWidth);

FPA fpStacked = ParallelArrays::Add(fpInput1,fpInput2);

FPA fpResult = ParallelArrays::Divide(fpStacked,2.0);

HANDLE evalCompleteEvent = CreateEvent(NULL, false, false, NULL);

DWORD nStart = GetTickCount();

evalTarget.ToArray(fpResult, \*stackedArray, arrayHeight, arrayWidth, arrayWidth\*sizeof(float), evalCompleteEvent);

DWORD nReturn = GetTickCount();

DWORD dwRet = WaitForSingleObject(evalCompleteEvent, INFINITE);

evalTarget.Delete();

DWORD nComplete = GetTickCount();

printf("Return time: %d Completion Time: %d\n",(nReturn - nStart), (nComplete - nStart));

for(int i = 0; i< 10;i++)

{

printf("[%d, 0] : %f\n",i, stackedArray[i][0]);

}

return 0;

}

# Appendix C: Complete Code for BoxCar2

The following is a complete listing of the BoxCar2 sample application. To run the sample, create a C# Windows console application and overwrite the code in program.cs with the code from the listing. For more details on how to set up .NET applications, see “Accelerator Programming Fundamentals” earlier in this paper.

using System;

using System.Diagnostics;

using System.Threading;

using Microsoft.ParallelArrays;

using FPA = Microsoft.ParallelArrays.FloatParallelArray;

using FPAP = Microsoft.ParallelArrays.FloatParallelArrayParam;

using PA = Microsoft.ParallelArrays.ParallelArrays;

namespace Boxcar2

{

class Boxcar2

{

static int evalCount;

static object completionLock;

static void Main(string[] args)

{

int arrayLength = 400000;

int numArrays = 10;

int i, j;

bool waiting = true;

Random ranf = new Random();

float[][] inputArrays = new float[numArrays][];

float[][] outputArrays = new float[arrayLength][];

FPA[] fpInputs = new FPA[numArrays];

FPA fpResult = new FPA(0, arrayLength);

FPAP fpInputParam = new FPAP("input array");

int filterWidth = 10;

Stopwatch evalTime = new Stopwatch();

float[] kernel = new float[filterWidth];

evalCount = numArrays;

completionLock = new object();

X64MulticoreTarget mcTarget = new X64MulticoreTarget();

AsyncCallback callback = new AsyncCallback(MCCallback);

for (i = 0; i < numArrays; i++)

{

inputArrays[i] = new float[arrayLength];

for (j = 0; j < arrayLength; j++)

{

inputArrays[i][j] = (float)(Math.Sin((double)j / 10.0) + ranf.NextDouble() / 5.0);

}

fpInputs[i] = new FPA(inputArrays[i]);

}

for (i = 0; i < filterWidth; i++)

{

kernel[i] = 1.0f / filterWidth;

}

for (i = 0; i < kernel.Length; i++)

{

fpResult += PA.Shift(fpInputParam, i) \* kernel[i];

}

evalTime.Start();

for (i = 0; i < numArrays; i++)

{

fpInputParam.Bind(fpInputs[i]);

mcTarget.BeginToArray(fpResult, out outputArrays[i], mcTarget, callback, ExecutionMode.ExecutionModeNormal);

}

Console.WriteLine("Evaluations started: {0}", evalTime.ElapsedMilliseconds);

while (waiting)

{

Thread.Sleep(50); //Placeholder for UI work.

lock (completionLock)

{

waiting = (evalCount > 0);

}

}

Console.WriteLine("Evaluations finished: {0}", evalTime.ElapsedMilliseconds);

for (i = 0; i <= 10; i++)

{

Console.WriteLine("OutputArray [{0}]: {1}", i, outputArrays[0][i]);

}

Console.ReadKey();

}

static void MCCallback(IAsyncResult result)

{

lock (completionLock)

{

evalCount--;

}

}

}

}

# Appendix D: Programming FAQ

This section contains answers to frequently asked questions from Accelerator programmers.

## Appending Arrays

It is common to need to glue two arrays together to make a new array. For example, it is possible to take a 3x5 array *a1* and a 3x7 array *a2* and glue them together to get a 3x12 array. This is accomplished with the following code:

FPA a1Pad = A.Pad(a1, new int[]{0, 0}, new int[]{0, 7}, 0.0f);

FPA a2Pad = A.Pad(a2, new int[]{0, 5}, new int[]{0, 0}, 0.0f);

FPA gluedA = a1Pad + a2Pad;

At first glance, this may seem inefficient. It appears as though it is a request to create two new larger arrays and then add them together. Doing that would require extra memory allocations along with extra reads and writes. However, when generating code, Accelerator will not create the padded arrays, it will generate code that reads from a1 or a2 based on the index of the element being fetched.