An Introduction to Microsoft Accelerator v2

Preview Draft #2 - Version 2.1 – June 16, 2010

Abstract

Microsoft® Accelerator v2 provides an effective way for applications to implement array-processing operations using the parallel processing capabilities of multi-processor computers. The Accelerator application programming interface (API) supports a functional programming model for implementing a wide variety of array-processing operations.

Accelerator handles all the details of parallelizing and running the computation on the selected target processor, including GPUs and multicore CPUs. The Accelerator API is almost completely processor independent, so the same array-processing code runs on any supported processor with only minor changes. This paper is a general introduction to Accelerator.

In this introduction

[About Accelerator v2](#_Toc264439083)

[Parallel Programming](#_Toc264439084)

[Accelerator Quick Start](#_Toc264439085)

[Accelerator Architecture](#_Toc264439086)

[The Accelerator Programming Model](#_Toc264439087)

[How Accelerator Applications Run](#_Toc264439088)

[Performance Considerations](#_Toc264439089)

[Resources](#_Toc264439090)

[Appendix A: How to Install Accelerator](#_Toc264439091)

[Appendix B: New Features under Consideration](#_Toc264439092)

[Appendix C: Source Code for the C++ Version of StackArrays](#_Toc264439093)

[Appendix D: StackMany Source Code](#_Toc264439094)

**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://connect.microsoft.com/acceleratorv2

**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 | First review draft | | | |
| November 10, 2009 | Preview content supporting the “parity” release of Accelerator vv2 | | | |
| June 16, 2010 | Preview #2. Updated to add coverage for new features: DoubleParallelArray, Evaluate, parameter objects, and asynchronous evaluation. | | | |

Contents

[About Accelerator v2 4](#_Toc264439095)

[Parallel Programming 5](#_Toc264439096)

[How to Implement a Parallelized Application 6](#_Toc264439097)

[Data-Parallel Programming and Array Processing 6](#_Toc264439098)

[How to Implement a Data-Parallel Operation 8](#_Toc264439099)

[Data-Parallel Programming with Accelerator 9](#_Toc264439100)

[Accelerator Quick Start 9](#_Toc264439101)

[The Accelerator Programming Pattern 11](#_Toc264439102)

[Key Accelerator Features 12](#_Toc264439103)

[Accelerator Architecture 13](#_Toc264439104)

[The Accelerator Library and APIs 13](#_Toc264439105)

[Targets 14](#_Toc264439106)

[Processor Hardware 14](#_Toc264439107)

[The Accelerator Programming Model 16](#_Toc264439108)

[Data-Parallel Array Objects 17](#_Toc264439109)

[Accelerator Operations 17](#_Toc264439110)

[The Target API 18](#_Toc264439111)

[Accelerator Expression Graphs 21](#_Toc264439112)

[The Evaluate Method 23](#_Toc264439113)

[Parameter Objects 24](#_Toc264439114)

[Asynchronous Evaluation 25](#_Toc264439115)

[How Accelerator Applications Run 26](#_Toc264439116)

[How StackArrays Runs 26](#_Toc264439117)

[A More Complex Accelerator Application 28](#_Toc264439118)

[StackMany on the MultiCore Target 32](#_Toc264439119)

[Performance Considerations 33](#_Toc264439120)

[Data Transfer Rate 33](#_Toc264439121)

[Input Data Size 33](#_Toc264439122)

[Number of Processors 34](#_Toc264439123)

[Operation-Related Issues 34](#_Toc264439124)

[Precision 35](#_Toc264439125)

[Resources 35](#_Toc264439126)

[Appendix A: How to Install Accelerator 36](#_Toc264439127)

[Appendix B: New Features under Consideration 37](#_Toc264439128)

[Targets 37](#_Toc264439129)

[Special-Purpose Operations 37](#_Toc264439130)

[Sets of Data-Parallel Array Objects 37](#_Toc264439131)

[Target API 38](#_Toc264439132)

[Appendix C: Source Code for the C++ Version of StackArrays 39](#_Toc264439133)

[Appendix D: StackMany Source Code 40](#_Toc264439134)

# About Accelerator v2

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 might not perform much better than on a single-processor system with the same clock speed. To take full advantage of multiprocessor systems, you usually must rewrite the application using parallel-programming techniques.

Microsoft Accelerator v2 provides an effective way for applications to implement array-processing operations using the parallel processing capabilities of multiprocessor computers. You use the Accelerator API and functional programming model to implement array-processing operations. Accelerator handles all the details of parallelizing the code and running the computation on the selected target processor, including GPUs and multicore CPUs. The Accelerator API is almost completely processor independent, so you can run the same array processing code on any supported processor with only minor changes.

This paper is a general introduction to Accelerator. For a detailed discussion of how to use Accelerator in applications, see “Microsoft Accelerator v2 Programming Guide” on the Accelerator Web site.

**Note:** This paper documents the Accelerator v2 Preview #2 release. For a discussion of additional features that are under consideration for the final Accelerator v2 release, see Appendix B.

Terminology

This section defines some terms that are either not in common use or are used in this document in Accelerator-specific ways.

data-parallel array object

An Accelerator object that represents an array in the Accelerator environment.

evaluate/evaluation

The process of evaluating the current state of a data-parallel array object and converting the data into a System object.

expression graph

A directed acyclic graph (DAG) used by Accelerator to represent a series of operations and the associated data.

shader

A program that runs on the GPU.

texture

A data structure that contains bitmap data to be applied to graphics surfaces. Accelerator uses textures to hold array data.

# Parallel Programming

Before discussing the specifics of Accelerator, it’s useful to briefly discuss “sequential” and “parallel” programming in general.

A sequential program consists of a sequence of instructions that are executed one at a time. Originally, all programs were sequential, and the model is still widely used. In practice, any program that runs on a single processor—even a modern multi-threaded application—is sequential in the sense that the processor can run only one instruction at time.

Parallel programming is a more recent development that takes advantage of systems with multiple processors. It is based on the recognition that programs—or at least key parts of programs—can often be divided into largely independent components. Each component can then be run on one of the available processors “in parallel” with other components running on other processors. Instead of one instruction at a time, a parallelized program can concurrently run as many instructions as there are processors, so it usually runs much faster than the sequential equivalent.

Modern client applications are typically implemented to run on a single processor system. If you run such an application on a multicore system, it will probably spend most much of its time running on a single processor, especially if most of the application’s work is handled by a single thread. The application is still effectively sequential and derives relatively little benefit from access to multiple processors. To use multiple processors effectively, the application—or at least its computationally intensive parts—must be explicitly parallelized.

Some applications are better suited to parallelization than others:

* Applications that spend most of their time performing computationally intensive tasks such as data mining or numerical modeling are often good candidates for parallelization.
* Applications that spend most of their time performing inherently sequential tasks, or are idle much of the time waiting for user input, might not benefit much from parallelization

Accelerator supports array processing, which is usually well-suited to parallelization.

## How to Implement a Parallelized Application

To parallelize an application, you must divide the key parts of the application into separate components and execute them in parallel on separate processors. There are two basic approaches:

Task Parallel

This approach focuses on parallelizing different programming tasks, and is typically used for applications that have a diverse set of loosely coupled tasks. The application assigns different tasks to different processors. Each task might share data with other tasks, but is otherwise independent.

Data Parallel

This approach focuses on parallelizing data, and is typically used for applications that need to process a large data set. The application divides the data for a task into multiple partitions and assigns the partitions to different processors. Each processor then runs the same code on its assigned data partition, and the application reassembles the processed partitions into a final result.

“Embarrassingly” data-parallel programs have no shared data. Each processor uses data only from its assigned data partition and runs independently of the other processors.

Parallel programming is not limited to multicore CPUs or even a single computer.

* Parallelized applications can run on a variety other processor types, such as GPUs and field-programmable gate arrays (FPGAs).
* Distributed computing technologies such as Dryad run parallelized applications on clusters of as many as several thousand of separate computers.

Accelerator runs locally, so this paper focuses on parallel programming for a single computer. For more information on distributed computing and Dryad, see “Dryad and DryadLINQ for Data Intensive Research,” which is listed in “Resources” at the end of this paper.

## Data-Parallel Programming and Array Processing

Array processing applications are good candidates for a data-parallel approach, and often for an embarrassingly data-parallel approach. Examples of array-processing operations that are well-suited for data-parallel programming include:

* Processing images, including operations such as rotation, blurring, or color filtering.
* Processing audio or video streams, including operations such as noise reduction, rotation, special effects, or merging streams.
* Processing scientific data, including tasks such as processing seismic reflection data or solving differential equations.

The best way to explain data-parallel array processing is with an example. A convenient operation for this purpose is element-wise addition, which is used for tasks such as “stacking” time series to improve the signal/noise ratio.

Element-wise addition adds each pair of elements from two numerical arrays, and yields a result array of the same length containing the sums. In a standard sequential application, the code to implement the operation would look something like the following example, where A and B are the arrays to be added, R is the result array, and N is the number of elements:

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

{

R[i] = A[i] + B[i];

}

On a single-processor system, the loop’s N iterations are computed one after the other. On a multicore system, the thread manager might run different iterations on different cores, but the iterations still run one at a time.

Data-parallel programming can often improve performance substantially. However, the operation must satisfy the following criteria:

* The operation’s input data can be partitioned into independent subsets.
* The operation can process each data partition separately, with relatively little shared data.

For embarrassingly data-parallel programs, there must be no shared data.

Element-wise addition satisfies both criteria, as do many array processing operations.

Figure 1 is a schematic diagram that shows both sequential and parallel approaches to element-wise addition. It assumes that the sequential version runs on a single processor system, and the data-parallel version runs on a multi-processor system with M processors (P1 - PM).



Figure 1. Sequential versus data-parallel programs

The data parallel version:

1. Divides the two input arrays into M partitions (A1 - AM and B1 - BM), each containing approximately N/M elements.

2. Assigns each pair of input partitions to the corresponding processor.

3. Each processor runs the element-wise addition code on its assigned data partitions, which produces a set of output partitions (R1 - RM).

The code running on each processor is identical to that used by the sequential program, except that it processes only the data in the assigned input partitions.

4. Reassembles R1 - RM in the correct order to produce the result array.

The data-parallel version of the application has to handle tasks such as partitioning and reassembling the data that aren’t required for the sequential version, but it should still run nearly M-times as fast as the sequential version.

Because the parallel processing code uses data only from the local input partitions, there is no shared data and the operation is embarrassingly data parallel.

Other array processing operations can be implemented as data-parallel programs, but not embarrassingly data parallel. For example, consider a reduction operation that sums all the elements of an array and yields a single number. The result depends on the entire array, so it is not embarrassingly data parallel. Reduction can still be implemented as a data-parallel operation, but the details are somewhat more complex than Figure 1. For example, you could have each processor sum its data partition, and then sum the results.

## How to Implement a Data-Parallel Operation

Data-parallel programming is simple in concept, but difficult in practice. To implement an operation like that described in Figure 1, the application must:

* Partition the input data and assign each partition to the appropriate processor.
* Generate the kernel, which is the code that executes the operation.
* Run the kernel on each processor to process the assigned input partition—synchronizing access to any shared data—and produce an output partition.
* When all kernels have finished, reassemble the results from each processor into a final result.

This can be a non-trivial problem even for a multicore CPU. With other processor types—such as GPUs or FPGAs—you must also translate the code and data into an appropriate form for the particular processor before you can run the operation.

For example, the following procedure is a general description of how to run a data-parallel application on a GPU by using Microsoft DirectX® 9. Other processors have different requirements, but the general approach is similar.

1. Translate the processing code into a form suitable for a GPU by converting it to a DirectX 9 pixel shader.

2. Translate the data into a format that is suitable for a GPU by converting it to a DirectX 9 texture.

3. Transfer the shader and textures to the processor and run the operation.

DirectX 9 and the associated drivers partition the data and schedule execution on the various pixel shaders. With other processors, the application might have to handle some or all of these tasks.

4. When the operation is complete, retrieve the texture containing the assembled results and convert it back to an array.

The program is specific to DirectX 9-compatible GPUs. To run this application on a different type of processor, such as an FPGA or digital signal processor (DSP), you must implement the operation separately for each processor type.

## Data-Parallel Programming with Accelerator

Although incorporating data-parallel programming into an application is straightforward in principle, it has been difficult in practice. For example, the GPU and its supporting APIs are highly customized to support graphics programming. To implement a GPGPU-based application, you have had to learn specialized graphics APIs such as DirectX or OpenGL plus specialized shader languages to program the GPU. You must then adapt the graphics APIs, languages, and data formats to the requirements of general-purpose computing. Some higher-level abstractions are available, but they still require developers to interact directly with the GPU.

Accelerator is a library that handles nearly all of the complications of implementing array processing operations as data-parallel programs. The Accelerator API supports a functional programming model that you use to implement your array-processing operation. Accelerator handles the details of running the operation as a data-parallel program.

Most of the Accelerator API—including all the array-processing operations—is processor independent. The only processor-specific code in most Accelerator applications is a standard method which directs Accelerator to evaluate the results of the operation on a specified processor. Accelerator then parallelizes the operation, runs it on the processor, and returns the result. You can usually run the same array-processing code on a different processor simply by calling a different evaluation method.

The remainder of this paper is a general discussion of Accelerator and how it supports parallelized array-processing applications. For a detailed discussion of how to implement Accelerator applications, see “Microsoft Accelerator v2 Programming Guide.” For instructions on how to install Accelerator, see Appendix A.

# Accelerator Quick Start

Before starting in on the details, it’s useful to examine how a simple Accelerator program works. The following example is an Accelerator application, StackArrays, that implements the element-wise addition operation discussed in the preceding sections.

The example in Listing 1 stacks two noisy 100-element sine waves and then normalizes the result by dividing it by 2. These are much smaller arrays than you would use in a real application, but it keeps the output manageable. The numbered comments are used in the following sections to identify the key parts of the code.

Listing 1: StackArrays

using System;

using Microsoft.ParallelArrays;

using FPA = Microsoft.ParallelArrays.FloatParallelArray;

using PA = Microsoft.ParallelArrays.ParallelArrays;

namespace AddArrays

{

class Program

{

static void Main(string[] args)

{

int arrayLength = 100;

Random ranf = new Random();

float[] inputArray1 = new float[arrayLength];

float[] inputArray2 = new float[arrayLength];

float[] stackedArray = new float[arrayLength];

// [1]

DX9Target evalTarget = new DX9Target();

// [2]

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

{

inputArray1[i] = (float)(Math.Sin((double)i / 10.0)

+ ranf.NextDouble() / 5.0);

inputArray2[i] = (float)(Math.Sin((double)i / 10.0)

+ ranf.NextDouble() / 5.0);

}

// [3]

FPA fpInput1 = new FPA(inputArray1);

FPA fpInput2 = new FPA(inputArray2);

// [4]

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

FPA fpOutput = PA.Divide(fpStacked, 2);

// [5]

stackedArray = evalTarget.ToArray1D(fpOutput);

// [6]

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

{

Console.WriteLine(stackedArray[i].ToString());

}

}

}

}

This example uses standard aliases to represent two commonly used types:

* **PA** represents **ParallelArrays**, which contains the Accelerator operation methods.
* **FPA** represents the **FloatParallelArray**, which represents floating point arrays in the Accelerator environment.

If you have installed Accelerator, you can run StackArrays as follows. The procedure assumes that you are building a debug version.

To build and run StackArrays

1. In Microsoft Visual Studio® 2008 or later version, create a new .NET Framework console application.

2. Open program.cs and replace the contents with the code from Listing 1.

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

The DLL is under the Program Files\Microsoft\Accelerator v2\bin\Managed folder. The Managed folder contains Debug and Release folders for the debug and release versions of the DLL.

4. Build the application.

5. Copy Accelerator.dll to the project’s bin\debug folder.

Accelerator.dll is under the Program Files\Microsoft\Accelerator v2\bin folder. 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. Press CTRL+F5 to run the application.

**Note:** You can also implement Accelerator operations with unmanaged C++. For a C++ version of StackArrays, see Appendix C.

## The Accelerator Programming Pattern

Although quite simple, StackArrays demonstrates the general programming pattern used by most Accelerator applications. The items in this list are keyed to the numbered comments in Listing 1.

1. Create a target object.

Each processor that supports Accelerator has one or more target objects, which convert Accelerator operations to processor-specific code and data and run the operations on the processor.

StackArrays uses **DX9Target**, which runs Accelerator operations on any DirectX 9-compatible GPU, using the DirectX 9 API.

2. Create input data arrays.

For simplicity, the input arrays for StackArrays are generated by the application, but they would typically be created from stored data.

3. Load each input array into an Accelerator data-parallel array object.

A data-parallel array object, such as **FloatParallelArray**, represents a data array in the Accelerator environment.

4. Use one or more Accelerator operations to process the arrays.

StackArrays calls two Accelerator methods:

**Add** performs element-wise addition on the two input arrays.

**Divide** normalizes the result by dividing each element by 2.0.

5. Evaluate the results of the series of operations by passing the result object to the target’s **ToArray1D** method.

**ToArray1D** method directs the target to determine the result object’s current state by running the associated operations as a data-parallel program on the GPU. A similar method, **ToArray2D**, evaluates 2-D arrays.

6. Process the result array further.

StackArrays displays the elements of the processed array in the console window.

## Key Accelerator Features

StackArrays illustrates several key Accelerator features.

Pluggable Target Objects

You can run Accelerator applications on a variety of processors by using the appropriate target object, each of which handles the details of evaluating Accelerator operations on a particular processor. Accelerator targets are pluggable, so IHVs or other third-party vendors can implement target objects for any suitable processor.

The **ToArray1D** call is one of only two lines of processor-specific code in StackArrays—the other is the line that creates the target object. All targets expose **ToArray1D**, so you can run StackArrays on a different target by creating an appropriate target object, and passing *fpOutput* to the object’s **ToArray1D** method.

Functional Programming Language

The Accelerator API supports a functional programming language for performing array processing operations, which aids parallelization and eliminates side effects. The input arrays do not change; each operation returns a new array. As an additional benefit, Accelerator code is typically much simpler and easier to create and maintain than conventional array-processing code. Instead of using **for** loops to iterate over array indices, StackArrays simply applies **Add** and **Divide** operations to the data-parallel array objects that represent the input arrays.

Implicit Parallelization

You do not have to handle any of the details of parallelizing your operations. Accelerator and the target object parallelize the computation and run it on the specified processor.

Processor-Agnostic

Most of the Accelerator API—including all the array-processing operations—is processor-agnostic. To evaluate a series of Accelerator operations on a different processor, you usually need to change only the line that creates the target object and the line that calls the evaluation method.

Deferred Execution

Accelerator does not perform the computation until StackArrays calls **ToArray1D** to evaluate the result object. Accelerator records the **Add** and **Divide** operations, but defers execution until the application explicitly directs the target to evaluate the final result. You could apply any number of additional operations to the output of **Divide**, and Accelerator would simply record the details, pending evaluation of a final result object.

Deferred execution allows you to perform an entire series of related operations on the target, which is usually more efficient than running one operation at a time.

# Accelerator Architecture

Figure 2 is a block diagram of the Accelerator v2 architecture.



Figure 2. Accelerator architecture

The following sections describe the various components of the Accelerator stack, and how they interact.

## The Accelerator Library and APIs

Applications use the Accelerator library to implement the processor-independent aspects of Accelerator programming, which constitute the majority of most Accelerator programs. The upper edge of the library exposes two APIs:

* C++ applications use the native API, which is implemented as unmanaged C++ classes.
* Managed applications use the managed API, which is a thin wrapper over the native API.

The lower edge of the library communicates with the Accelerator targets.

Both APIs support the same functionality, and the method syntax is as close as possible. For more details about the APIs and how to use them in applications, see “Microsoft Accelerator v2 Programming Guide.”

The performance difference between the two libraries is usually negligible. The managed API is a very lightweight wrapper over the C++ API, and does very little processing. The underlying C++ library and the target objects handle almost all of the computational workload.

In general, the choice is usually a matter of convenience.

* The managed API is usually preferable for a new application, especially if you want to implement a user interface.
* The C++ API is usually preferable if want to integrate Accelerator into an existing C++ application.

## Targets

Accelerator depends on a set of pluggable target objects. Each target translates Accelerator operations and data objects into a suitable form for its associated processor and runs the computation on the processor. A processor can have multiple targets, each of which accesses the processor in a different way. For example, most GPUs will probably have DirectX 9 and DirectX 11 targets, and possibly a vendor-implemented target to support processor-specific technologies.

Each target exposes a small API, which applications call to interact directly with the target. The most commonly used methods are **ToArray1D** and **ToArray2D**, which direct the target to evaluate the results of a series of Accelerator operations on the target’s processor. Targets can optionally expose additional methods, as discussed later in Appendix A.

To run parallelized code efficiently, a target must include a resource manager that balances the processing load across the target’s individual processors:

* Accelerator’s multicore CPU target supports pluggable resource managers.
* DirectX targets handle resource management internally.

Third-party targets can handle resource management in any way that they prefer. Refer to the specific target documentation for details.

## Processor Hardware

Accelerator v2 can run operations on a variety of targets. Accelerator currently includes targets for:

* Multicore x64 CPUs.
* GPUs, by using DirectX 9.

Additional targets—including support for GPUs by using DirectX 11 (DirectCompute)—might be available in the future.

Accelerator v2 can run operations on other suitable processor types such as digital signal processors or FPGAs, if a target is available. Those targets must be provided by third-party vendors.

### How Accelerator Runs Operations on a GPU

To understand how Accelerator performs calculations on the GPU, it is useful to first look at the GPU architecture. Figure 3 shows a simplified block diagram of a typical GPU.



Figure 3. Block diagram of a GPU

Graphics images are composed of surfaces whose shapes are defined by a mesh of linked triangles. The mesh is usually wrapped by a texture, which adds color and fine details to the basic structure defined by the mesh.

Accelerator’s DirectX GPU targets are based on shaders, which are small programs written for the GPU:

* Vertex shaders run on the vertex processors.

They process the vertices that define the triangles that make up the mesh, and perform tasks such as transforming the position or orientation of an object.

* Pixel shaders run on the pixel processors.

They compute the values of the output pixels and handle tasks such as lighting effects and applying textures to the mesh. Pixel shaders have easy access to the textures stored in GPU memory

Accelerator performs operations on the GPU by using textures and pixel shaders:

* Accelerator translates the operation’s data into textures.
* Accelerator translates the operation’s programming logic into pixel shader code.

Accelerator programs the GPU by using the DirectX API, which is supported by a wide range of GPUs.

The pixel shaders process the textures and Accelerator translates the results into arrays and returns the arrays to the application.

### How Accelerator Runs Operations on a Multicore CPU

The multicore target runs Accelerator operations on the system’s CPUs and uses system memory. To run Accelerator operations, the target:

Copies the input data to another memory location

The target works with a copy of the input data to eliminate the possibility of a race condition with the application.

Creates a set of threads

The target’s resource manager determines the number of threads, based on the input data size and the number of cores. For relatively small input data sets, the target might create fewer threads than cores. In some cases, the target might create more threads than cores, which allows the computation to take advantage of any cycles that aren’t being used by the other threads.

Spawns one instance of the kernel for each thread

Each thread runs the same kernel on its assigned data partition.

Runs the kernels

The thread manager in the Windows® operating system determines which processor each kernel runs on. The kernels must compete for resources with various operating system threads, plus whatever applications might be running. Letting the thread manager handle thread assignment ensures optimal use of CPU resources.

Reassembles the results and returns it to the application

After all the threads have completed, the target reassembles the results from each thread into a result array and returns it to the application. The target does not copy the result array. The application doesn’t have access to it until evaluation is complete, so there is no risk of a race condition.

# The Accelerator Programming Model

As discussed earlier, the basic Accelerator programming pattern is:

1. Create data arrays.

2. Load the data arrays into Accelerator data-parallel array objects.

3. Use a series of Accelerator operations to process the data-parallel array objects and create a result object.

4. Evaluate the result object on a target.

This section discusses how the model works in more detail.

## Data-Parallel Array Objects

Data-parallel array objects are Accelerator’s fundamental data type. Accelerator v2 supports five data-parallel array objects:

BoolParallelArray  
DoubleParallelArray   
Float4ParallelArray   
FloatParallelArray   
IntParallelArray

A data-parallel array object represents a single rank one or rank two array—often referred to as a one-dimensional or two-dimensional array—of the respective primitive type: **bool**, **double**, **Float4, float**, and **int**.

**Note:** **Float4** is an Accelerator structure that contains a quadruplet of **float** values. It is used primarily in graphics programming.

Data-parallel array objects are largely opaque to applications. In particular, applications do not have direct access to the array data and cannot manipulate the array by index. Applications implement array processing schemes by applying Accelerator operations to data-parallel array objects—which represent an entire array as a unit—rather than working with individual array elements.

## Accelerator Operations

The Accelerator API supports a functional programming language that handles most common array processing procedures. The API includes a large collection of operations that applications can use to manipulate the contents of arrays and to combine the arrays in various ways.

Most operations take one or more input data-parallel array objects, and return the processed data as a new data-parallel object. The **Add** and **Divide** operations used by StackArrays are typical examples. The exceptions are a small number of operations such as **Positions**—known as procedural operations—which generate their output internally, and take no input. Accelerator operations work with copies of the input objects; they do not modify the original objects.

Operation inputs are typically data-parallel objects, but some operations can also take constant values. A constant is treated as a data-parallel object that represents an array of the appropriate dimensions with all elements set to the specified constant. For example, if you replace *fpInput1* in StackArrays with v2, **Add** adds 2.0 to each element of *fpInput2*.

Accelerator exposes operations as a C++ functions and .NET methods:

* The .NET operations are implemented as static methods in the **Microsoft.ParallelArrays.ParallelArrays** type, 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.

The operation names are the same for both APIs, and the syntax and usage are as similar as possible. 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.

Table 1 summarizes the available Accelerator operations. For a detailed list, see the Accelerator Help file (Accelerator.chm), which is in the Program Files\Microsoft\Accelerator\doc folder.

Table 1. General Categories of Accelerator Operations

|  |  |
| --- | --- |
| Operation | Description |
| Creation and conversion | Create new data-parallel array objects and convert them from one type to another. |
| Element-wise | Operate on each element of one or more data-parallel array objects and return a new object with the same dimensions as the originals. For example:   * **Add** sums each pair of elements from two objects and returns an object containing the sums. * **Abs** determines the absolute value of each element of an object, and returns an object containing the absolute values. |
| Reduction | Reduce the rank of a data-parallel array object by applying a function across one or more dimensions. For example:   * One overload of **Sum** sums the elements of each row of a rank two object, which yields a rank one object that contains the sums. * Another overload of **Sum** sums every element in the array and returns the sum. |
| Transform | Transform the organization of the elements in a data-parallel array object. These operations reorganize the data in the object, but do not require computation. For example:   * **Transpose** performs matrix transposition on rank two objects. * **Pad** increases the size of an object by adding new elements. |
| Linear algebra | Perform standard matrix operations on data-parallel array objects, including matrix multiplication, scalar product, and outer product. |

**Note:** Several additional operations are under consideration for the Accelerator v2 final release. For details, see Appendix B.

## The Target API

Accelerator targets expose a primary interface and can optionally expose additional methods.

### The Primary Target Interface

Accelerator targets expose a standard C++ interface and an associated managed wrapper, which supports the following:

Object creation.

The C++ API supports target object creation through either a function or a static method on the target class. For example, the DirectX 9 target exposes an object creation function, **CreateDX9Target**. Targets can optionally expose multiple object creation functions or methods. For example, the multicore target exposes four object creation functions, which allow you so specify such factors as which pluggable resource manager to use.

Managed applications create a target object by applying the **new** operator to the appropriate target class. For example, the DirectX 9 target object is implemented by the **DX9Target** class. Targets with multiple object-creation functions typically implement separate constructors for each creation function, but that detail is up to the target implementer.

Object destruction.

The C++ target type exposes an object destruction method, **Delete**. The .NET garbage collector handles object destruction for managed target objects.

Evaluation

The bulk of the target object’s API is a set evaluation methods—referred to collectively as **ToArray**—which direct the target to evaluate a data-parallel array object, and return an array of the appropriate type and rank.

They are overloaded to accommodate the various data-parallel array object types and ranks.The evaluation methods also have separate sets of overloads for synchronous and asynchronous evaluation, which is discussed later.

If a processor cannot support a particular object type, the target might choose to provide only minimal implementations of some of the **ToArray** overloads. They 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 the **ToArray** overloads that evaluate **IntParallelArray** objects.

**Note:** Most applications use **ToArray1D** and **ToArray2D**, which return the array that contains the processed data. Targets also implement a set of **ToArray** overloads that are actually named **ToArray**. They take a data-parallel array object and return the result array through an **out** parameter instead of as a return value. **ToArray1D** and **ToArray2D** are just wrappers for the corresponding **ToArray** methods.

### Target Memory Interface

Targets can expose an optional target memory interface, to allow applications to perform special-purpose operations outside the Accelerator framework. This interface is particularly useful for processors such as GPUs, where it typically takes a significant amount of time to transfer data from system memory to processor memory and back again.

**Note:** Because of the low-level nature of the target memory interface, it might not be supported by the target’s managed wrapper.

The target memory interface has two methods, which typically have multiple overloads to handle different data-parallel array object types. The names used here are placeholders; each target implementer defines the names and syntax for the methods that they implement:

*ToTargetMemory*

This method takes a data-parallel array object, evaluates it, and places the results in target memory in an appropriate format

*FromTargetMemory*

This method takes data from target memory, and returns it to the application as a data-parallel array object.

For example, the DirectX 9 target supports the target memory interface as follows:

* The ***ToTargetMemory*** implementation is named **ToTextureMemory**. It evaluates a **FloatParallelArray** or **Float4ParallelArray** object, converts the results to a texture, and transfers the texture to GPU memory.
* The ***FromTargetMemory*** implementations are named **FromFloatTextureMemory** and **FromFloat4TextureMemory**. They take a texture from GPU memory and return it as a **FloatParallelArray** or **Float4ParallelArray** object, respectively.

**Note:** The DirectX 9 target memory interface does not currently support **BoolParallelArray**.

The target memory interface is typically used by applications that have an existing library of processor-specific code, and want to incorporate Accelerator into the application without rewriting the library. For example, you might have a neural network application that already includes a library to perform specialized neural-network processing operations on a GPU, but would like to use Accelerator to perform additional processing.

The following two examples show how such an application would work on a GPU with and without a target memory interface.

Example: Without a target memory interface

1. The application uses Accelerator to prepare the initial data and calls **ToArray** to evaluate the resulting data-parallel array object.

2. The target evaluates the data-parallel array object on the GPU and returns the results to the CPU as an array.

3. The application converts the array into an appropriate format, such as a texture, and returns the data to the GPU for processing.

4. The application uses the neural network library to process the data.

5. The application retrieves the data from the GPU and packages it as an Accelerator object.

6. The application performs further Accelerator operations on the result object and calls **ToArray** to evaluate the final result.

7. The target evaluates the second set of Accelerator operations on the GPU and returns the result to the CPU, and to the application.

This approach requires three CPU-GPU roundtrips, which is very time-consuming. If the target exposes a target memory interface, the application can integrate the neural network library with Accelerator much more efficiently.

The following example shows how the process works with the DirectX 9 target.

Example: With the DirectX 9 target memory interface

1. The application uses Accelerator to prepare the initial data and passes the data-parallel array object to **ToTextureMemory**.

2. **ToTextureMemory** evaluates the Accelerator object on the GPU and stores the results in GPU memory as a texture.

**ToTextureMemory** returns a GPU memory pointer to the application.

3. The application uses its neural-network library to process the results from Step 2 on the GPU and stores the results in GPU memory.

4. After processing is complete, the application calls **FromFloatTextureMemory** or **FromFloat4TextureMemory** to retrieve the results from GPU memory, packaged as a data-parallel array object.

These methods do not transfer any data to the CPU. It remains on the GPU pending any further Accelerator operations.

5. The application uses Accelerator to process the data further and calls **ToArray** to evaluate the final result.

6. The target runs the remaining operations on the GPU, using the stored data from Step 3, and returns processed data to the CPU, and to the application.

The data to be processed is thus transferred to the GPU in Step 1 and remains there until it is returned to the CPU in Step 6. This eliminates two CPU-GPU round trips compared to the first example, which can significantly improve performance.

## Accelerator Expression Graphs

As an application progresses through a series of operations, Accelerator does not immediately perform any computations. Instead, Accelerator defers execution and records the details in a DAG—called an expression graph—that represents the operations’ programming logic and associated data. Accelerator stores the graph in system memory and continues to add to it until the application calls **ToArray**. The target then uses the expression graph to perform the computations required to evaluate the operations’ result object.

Applications do not interact directly with expression graphs. Although the Accelerator code that precedes an evaluation is often referred to as “expression graph construction,” the Accelerator library actually constructs and manages the expression graph, based on the application’s code. However, you should have a general understanding of the relationship between Accelerator operations and the associated expression graphs. This section provides a brief introduction. For a detailed discussion of expression graphs and how targets use them during the evaluation process, see “Microsoft Accelerator Target Implementers’ Guide.”

Figure 4 shows the expression graph for StackArrays.



Figure 4. StackArrays expression graph

The graph in Figure 4 is quite simple, but shows the essential features. Expression graphs are composed of nodes and links:

* Data nodes (unshaded) represent input data.

Data nodes represent a constant or a data-parallel array object. They have only output, which is linked to at least one operation node’s input. A data node can be linked to multiple operation nodes as long as the graph is acyclic, which means that it does not contain any closed paths.

* Operation nodes (shaded) represent Accelerator operations.

The operation node is usually linked to one or more input nodes, which can be either data nodes or the output of previous operation nodes. The node’s output is usually linked to another operation in the series. The exception is procedural operation nodes, which generate their output internally and have no input nodes.

The graph’s root node—**Divide** in this example—is at the top, and represents the operation that yields the result object, *fpOutput*, that is being evaluated. The remainder of the graph represents the operations and associated data that determine the *fpOutput* object’s current state.

To interpret an expression graph, start from the bottom and work up. For Figure 4:

1. **Add** operates on two input data nodes—*fpInput1* and *fpInput2*—both of which represent data-parallel array objects.

2. **Divide**, the root node, operates on the output of **Add** and a data node that represents a constant.

3. **Divide** yields the result object, *fpOutput*, which represents the result of the series of operations.

Every data-parallel array object has an associated expression graph, which is often a subset of a larger graph. For example, if the application had chosen to evaluate the result object produced by **Add**, the associated graph would consist only of **Add** and its input nodes.

When StackArrays calls **ToArray**, the target uses the expression graph in Figure 4 to determine the current state of *fpOutput*, as follows:

1. The target obtains the associated expression graph.

2. The target converts the operations and data represented by the graph into a form that can be efficiently executed on the target processor.

3. The target executes the operations on the target processor and returns the result to the application as an appropriately typed array.

For a detailed discussion of this process, see “Microsoft Accelerator Target Implementer’s Guide.”

## The Evaluate Method

Usually, you implement a series of Accelerator operations, call **ToArray**, and let the target determine how to handle evaluation. However, you can sometimes improve performance by including one or more **ParallelArrays.Evaluate** operations in the series. **Evaluate** explicitly directs the target to evaluate the expression graph at that particular point and use the stored result for further computations. The stored result is used only by the target and is not returned to application.

By default, targets usually attempt to improve performance by evaluating selected subsets of the expression graph—called common sub-expressions—and storing the result in processor memory for later use. In some cases, calling **Evaluate** has little or no effect on performance, because the target is already performing a similar action. **Evaluate** is useful for those cases where the target does not recognize the opportunity. For more information on how targets optimize expression graphs, see “Microsoft Accelerator Target Implementers’ Guide.”

There are no hard and fast rules for when to use **Evaluate**. If you think **Evaluate** might improve performance, you must determine whether or where to use it by experiment. Some examples of scenarios where **Evaluate** might improve performance include:

* When a result can be used multiple times later in the graph.

The target can then use a stored value instead of computing it each time.

* When the graph is too large for the target to handle efficiently.

This scenario is probably relatively rare, but could occur for large or complex computations, such as highly recursive computations.

**Important:** Use **Evaluate** sparingly. It doesn’t necessarily improve performance, and can actually degrade performance in some scenarios. For example, each **Evaluate** call requires the target to compute subexpressions, which involves reading from and writing to target memory. If you call **Evaluate** too frequently on a limited memory device such as a GPU, the reading and writing is expensive and can reduce performance.

The results of an **Evaluate** call are stored locally in target memory, so other targets do not have access to the value. If you evaluate the same series of operations on two targets, they compute the result of the **Evaluate** operation independently.

**Note:** With Accelerator v1.1, **Evaluate** initiated an evaluation process on the processor. However, the result remained in processor memory and nothing was returned until the application later called **ToArray**. Because Accelerator v2 must support multiple targets, it simply adds a node to the expression graph, and defers execution until you call **ToArray**. From an application perspective, **Evaluate** works much the same way in both Accelerator versions. All that has changed is the details of how the target handles the operation.

## Parameter Objects

Some array-processing scenarios run the same series of operations repeatedly, with different data. For example, you might want to perform the same series of signal-processing operations on a set of input arrays. The expression graph doesn’t change from one input array to the next; it just has different input data attached to a data node.

The following example shows how to implement this scenario by using data-parallel array objects:

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[i] = evalTarget.ToArray1D(fpFinalResult);

}

Each time you iterate this loop, Accelerator builds a new expression graph, even though it’s just the same graph with a different data-parallel array object attached to the first data node.

Parameter objects allow applications to treat expression graphs somewhat like functions. They can improve performance and simplify code for scenarios such as the preceding example. Parameter objects combined with asynchronous evaluation provide a very powerful and flexible way to evaluate Accelerator operations. For more discussion, see “Asynchronous Evaluation” later in this paper.

A parameter object is essentially a placeholder for a data-parallel array object. You use them with Accelerator operations in place of data-parallel array objects. Before calling **ToArray**, bind a data-parallel object of the appropriate type and dimensions to each parameter object, and then Accelerator attaches the data to the corresponding data nodes. To run the same operations again with different data, bind a new set of data-parallel array objects and call **ToArray** again. Instead of building a new graph, Accelerator simply attaches the new data to the appropriate data nodes.

With parameter objects, you could implement the preceding example something like the following:

fpInput = new FloatParallelArrayParams( );

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

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

...

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

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

{

fpInput.Bind(fpInputArrays[i]);

resultArray[i] = evalTarget.ToArray1D(fpFinalResult);

}

For a sample walkthrough, see “Microsoft Accelerator v2 Programming Guide.”

## Asynchronous Evaluation

With Accelerator 1.1, evaluation was always synchronous. For Accelerator v2, targets also support asynchronous evaluation, which allows your application to perform other tasks while it waits for the results of a time-consuming computation.

With asynchronous evaluation, the target initiates evaluation on a separate thread and returns immediately. When the computation is finished, the target populates the result array with the processed data and notifies you that evaluation is complete and the result array contains valid data. From an application perspective, the details are somewhat different, depending on whether you use the managed or unmanaged Accelerator API.

Managed. The managed API uses the standard .NET **BeginInvoke**/**EndInvoke** pattern. Accelerator implements these methods as **BeginToArray** and **EndToArray**. The following is a typical pattern:

1. To start evaluation, call **BeginToArray**.

**BeginToArray** returns an **IAcceleratorAsyncResult** interface, which is derived from **IAsyncResult**. You can use the interface to wait on the completion event, or you can pass **BeginToArray** a callback delegate, which is called when evaluation is complete.

2. When computation is complete, the target notifies the application by first calling the callback delegate—if one was provided—and then by signaling the completion event.

The target can also call **Target.EndToArray** at any time after **BeginToArray** returns. **EndToArray** is synchronous, and returns only after the evaluation is complete.

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

Unmanaged. The unmanaged API uses a set of **ToArray** overloads.

1. To start evaluation, call **ToArray** and pass it an event handle.

2. 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.

For more discussion of asynchronous evaluation, including walkthroughs of managed and unmanaged examples, see “Microsoft Accelerator v2Programming Guide.”

If you have multiple processors available—such as multiple graphics cards—you can improve performance by using asynchronous evaluation to implement a “fastest processor wins” model. Run the evaluation asynchronously on all available targets and wait on the results. When you receive the first notification, cancel the remaining evaluations and proceed.

Parameter objects improve performance by allowing you to re-use the same expression graph multiple times by simply binding new data-parallel array objects and running a new evaluation. You can further increase the power and flexibility of such applications by using asynchronous evaluation. Two examples of how to use this model are:

* Bind the data-parallel array objects and start asynchronous evaluations one after the other in a loop.

The target schedules the evaluations, and the application can then perform other tasks while waiting for all the evaluations to complete. For a simple example of this pattern, see “Sample Walkthrough: A Sliding-Window Filter Implemented with Parameter Objects” in “Microsoft Accelerator v2 Programming Guide.”

* If you have multiple processors, start different evaluations in parallel on different processors.

As each evaluation completes, bind a new set of data-parallel objects and start another one.

# How Accelerator Applications Run

Much of the code in a typical Accelerator application runs on the CPU as a normal Windows-based application. Only the Accelerator operations themselves run on the target, and only when the application evaluates a result object. This section describes how an Accelerator application runs. It uses a GPU target as an example, but the basic principles apply to all targets.

## How StackArrays Runs

This section discusses how a simple Accelerator application—the StackArrays example discussed earlier—runs on the GPU. The details are different for other processors, but follow the same general pattern.

The basic StackArrays logic is:

1. Create a target object.

2. Create data arrays.

3. Load the data arrays into data-parallel array objects.

4. Apply Accelerator **Add** and **Divide** operations to the data-parallel array objects.

5. Evaluate the result of the operations on the target.

6. Display the results.

Running StackArrays involves six participants:

* The application.
* The .NET Framework, which handles tasks such as creating the data arrays and displaying the results.
* The Accelerator library, which is used to process the data.
* The target object, which performs the computation on the target processor.
* The CPU, which runs most of the application.
* The target processor—a GPU in this example—which runs the array-processing computation.

Figure 5 shows how and where the various parts of StackArray run.



Figure 5. How StackArray runs

The code that runs on the CPU is determined at compile time. Its primary purpose is to construct the expression graph that defines the computation, and manage the logistics of running the computation on the target processor.

The code that actually performs the computation is created at run time by the target object and runs as an atomic block on the target processor. The target doesn’t return the results or control to StackArrays until the computation is complete.

**Note:** This chart shows synchronous evaluation, which doesn’t return control to the application until the evaluation is finished. Asynchronous evaluation immediately returns control to the application, and returns the results when the evaluation is complete. For more discussion of asynchronous evaluation, including a walkthrough of an application that is similar to StackArrays, see “Microsoft Accelerator v2 Programming Guide.”

## A More Complex Accelerator Application

StackArrays is a very simple application that stacks just two arrays, and involves only two Accelerator operations in sequence. Most applications are more complex and involve loops and other control structures. This section is based on a somewhat more complicated version of StackArrays, which uses a loop to stack multiple arrays. The basic program logic is:

1. Create a target object.

2. Create an array of data arrays, ten arrays in this example.

3. Load the data arrays into data-parallel array objects.

4. Stack the data-parallel array objects by adding each object to the sum of the preceding objects and normalizing the result.

5. Evaluate the results on the target.

6. Display the results.

There two basic ways to implement this application:

* Evaluate the result after each iteration.
* Evaluate the result after the entire operation is complete.

This section discusses both approaches, using different versions of StackMany. The examples are somewhat artificial, but they illustrate some key Accelerator performance issues which are discussed in the final section.

### StackMany1: Evaluate After Each Iteration

StackMany1 creates an array of 10 data arrays named *inputArrays*, each containing a noisy sine wave. It then stacks them by using the code shown in Listing 2. For the complete source code, see Appendix D.

Listing 2: StackMany1

static void Main(string[] args)

{

...

stackedArray = inputArrays[0];

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

{

fpStacked = new FPA(stackedArray);

fpInput = new FPA(inputArrays[i]);

fpOutput = PA.Add(fpStacked, fpInput);

stackedArray = evalTarget.ToArray1D(fpOutput);

}

fpStacked = new FPA(stackedArray);

fpOutput = PA.Divide(fpStacked, numArrays);

stackedArray = evalTarget.ToArray1D(fpOutput);

...

}

To stack the arrays, StackMany1:

1. Loads the first array into a **FloatParallelArray** object.

2. Loads the next array into a **FloatParallelArray** object.

3. Adds the two data-parallel array objects.

4. Evaluates the results, which yields a result array that contains the two stacked arrays.

5. Repeats Steps 1-4—using the result array from Step 4 as the first array in the next iteration—until all arrays have been processed.

6. Loads the final result array into a **FloatParallelArray** object and normalizes the result by applying **Divide**.

7. Evaluates the final result.

Figure 6 shows how this procedure would run by using a GPU target. It omits the target creation and results processing stages, which are similar to Figure 5.



Figure 6. How StackMany1 runs

The computation is based on the two simple expression graphs shown in Figure 7.



Figure 7. StackMany1 expression graphs

The graphs are used for different stages of the operation:

* The stacking computations are based on the Stacking graph.

It is used ten times, once for each iteration, with the result array from the previous iteration loaded into *fpStacked*.

* The final computation is based on the Normalization graph.

It is used once, to normalize the result array from the stacking operation and produce the final result.

### StackMany2: Evaluate After Stacking is Complete

StackMany2 is similar to StackMany1, but it stacks the arrays using the code shown in Listing 3. For the complete source code, see Appendix C.

Listing 3: StackMany2

...

fpStacked = new FPA(inputArrays[0]);

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

{

fpInput = new FPA(inputArrays[i]);

fpStacked = PA.Add(fpStacked, fpInput);

}

fpStacked = PA.Divide(fpStacked, numArrays);

stackedArray = evalTarget.ToArray1D(fpStacked);

...

To stack the arrays, StackMany2:

1. Loads the first array into a **FloatParallelArray** object.

2. Loads the next array into a **FloatParallelArray** object.

3. Adds the two data-parallel array objects, which yields a data-parallel array object that represents the stacked arrays.

4. Repeats Steps 2 and 3 until all data-parallel array objects have been processed.

Step 3 adds the data-parallel array object produced by the previous iteration to the object that represents the next array in the set.

5. Normalizes the result by applying **Divide** to the final data-parallel array object produced by Step 4.

6. Evaluates the final data-parallel array object from Step 5 to produce the final result array.

Figure 8 shows how this procedure would run by using a GPU target.



Figure 8. How StackMany2 runs

StackMany2 has only one evaluation step. During the operations that precede evaluation—including all iterations of the stacking operation—Accelerator is simply constructing the expression graph shown in Figure 9 and storing it in memory.



Figure 9. StackMany2 expression graphs

When StackMany2 calls **ToArray** to evaluate the final result, the target uses this graph to run the entire operation as an atomic computation on the GPU, and then returns the final result array. The StackMany2 evaluation process is significantly faster than StackMany1. The exact values depend on the particular computer and GPU, but on the computer used to write this paper, StackMany2 is approximately four times faster. The reasons for this performance difference are discussed later.

## StackMany on the MultiCore Target

If you run the StackMany operations on the multicore target instead of the GPU, it runs in much the same way. The primary differences are:

* The computations run on the CPU, not the GPU.
* The generated code is ordinary binary code, not shaders.
* The input data is ordinary data objects, not DirectX 9 textures.
* The input data is copied to another location in system memory, not across the PCIe bus to GPU memory.

Using a copy of the input data avoids the possibility of a race condition if the application were to modify the original data before evaluation is complete.

# Performance Considerations

Both versions of StackMany produce the same result on any target, so the primary distinction between them is performance. Accelerator performance is controlled in part by how you implement your application and in part by which target evaluates the results.

It is difficult to provide generally applicable performance guidelines; there are too many variables to consider. The most accurate way to evaluate performance is to experiment with different targets and application designs. This section describes some general performance considerations, and how they are affected by target characteristics and application design.

## Data Transfer Rate

To evaluate a data-parallel array object, the target must obtain the expression graph, convert the code and data to a suitable format, and run the operations on the processor. The largest single factor is often the time it takes to transfer the code and data to the target processor.

Target Choice

Data-transfer overhead is relatively high for GPU targets, because code and data for each evaluation must be transferred to and from the GPU over a relatively slow bus. The performance difference between StackMany1 and StackMany2 is primarily due to the fact that StackMany1 requires eleven CPU-GPU round trips and StackMany2 requires only one.

The multicore target uses the same processor and memory as the application proper, but it does copy the input data from one system memory location to another. This is much faster than transferring data to GPU memory, so multicore targets usually have less data-transfer overhead than GPU targets.

Application Design

In general, keep the number of evaluations as small as possible, but this factor is less critical for a multicore target. If the expression graph becomes very large, you might improve performance by introducing additional evaluations to produce a larger number of smaller graphs. However, graph size is not a performance issue for most applications.

## Input Data Size

The input data size can have a significant effect on performance and your choice of target.

Target Choice

To use the GPU effectively, you must use input objects that are reasonably large, but not too large.

With small arrays, the efficiencies of parallelized computation are more than offset by the cost of transferring the data to the GPU.

As a general rule, the minimum array size that runs efficiently on a GPU is approximately 106 elements. The exact figure depends on the particular GPU and the details of the computation.

GPU targets have a maximum array size.

For the DirectX 9 GPU target, one-dimensional array length is limited to the texture width and two-dimensional array dimensions are limited to width x height. For older video adapters, this means that 1-D arrays are limited to 4 thousand elements and 2-D arrays are limited to 16 million elements. With more recent video adapters, the limits are 8 thousand and 64 million elements.

The multicore target uses virtualized memory, like any other Windows-based application, so there is no effective limit on array size.

Application Design

For a GPU target, you must use sufficiently large arrays to ensure efficient computation without exceeding the size limit imposed by your GPU. If the input arrays are too large for the GPU, you might be able to subdivide the arrays and process them in stages.

The multicore target has no effective size limits, so it is often the best choice for one-dimensional arrays, and might be a better choice for very large two-dimensional arrays.

## Number of Processors

The number of processors controls how many concurrent operations can run on the target.

Target Choice

GPUs typically have 128 processors, many more than most multicore CPUs. The actual calculations thus typically run much faster on the GPU. Applications typically run faster on the GPU, as long as the arrays are large enough to compensate for the GPU’s high data-transfer overhead.

## Operation-Related Issues

There are several performance issues related to certain types of operations.

Reduction Operations

Reduction operations reduce the dimension of an array by combining rows or columns to produce an array of lower rank. For example, **Sum** adds the elements of each row of a two-dimensional array to produce a one-dimensional array that contains the sums or adds all elements of the array to produce a single value.

Reduction operations can be relatively slow on the GPU target, because it lacks accumulators and must perform the operation with multiple passes. Applications that use reduction operations extensively might perform better with the multicore target.

Transform Operations

Some transform operations, such as **Shift** or **Section**, can skip multiple elements, depending on their **Start** or **Stride** values. The multicore target operates on a cached subset of the input data. Large **Start** or **Stride** values can end up pointing beyond the cached data, which slows the computation. The GPU does not have this limitation.

## Precision

Accelerator provides two floating point data-parallel array objects, **FloatParallelArray** and **DoubleParallelArray**, which represent single-precision and double-precision floating point arrays, respectively. The two types can be used interchangeably with almost all Accelerator operations. However, using **DoubleParallelArray** can reduce evaluation performance relative to the same series of operations implemented with **FloatParallelArray**. For example, using **DoubleParallelArray** roughly doubles evaluation time on the multicore target. In general, use **FloatParallelArray** unless you require the higher precision of **DoubleParallelArray**.

**Note:** **DoubleParallelArray** is not supported by the DirectX 9 targets, so you cannot currently use it with GPUs.

# Resources

This section provides links to information about Accelerator and related topics.

#### 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/>

Microsoft Accelerator Updates and Software Availability News

<http://connect.microsoft.com/acceleratorv2>

Microsoft Research Accelerator Project Download

<http://research.microsoft.com/research/downloads/Details/25e1bea3-142e-4694-bde5-f0d44f9d8709/Details.aspx>

#### Related Resources

Calling Synchronous Methods Asynchronously

<http://msdn.microsoft.com/en-us/library/2e08f6yc.aspx>

Directed acyclic graph

<http://en.wikipedia.org/wiki/Directed_acyclic_graph>

Dryad and DryadLINQ for Data Intensive Research

<http://research.microsoft.com/en-us/collaboration/tools/dryad.aspx>

DirectX Developer Center

<http://msdn.microsoft.com/en-us/directx/default.aspx>

# Appendix A: How to Install Accelerator

You can obtain Accelerator from the Microsoft Connect Web site at <https://connect.microsoft.com/acceleratorv2>.

To install Accelerator

1. On the Microsoft Connect Web site for Accelerator v2, click the Download link 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.

# Appendix B: New Features under Consideration

This appendix discusses the new features that are under consideration for the final release of Accelerator v2, but are not included in the v1.1 parity release. This is a preliminary discussion, and features may be changed substantially prior to final release of Accelerator v2 software.

## Targets

Accelerator v2 might include two additional targets, either as part of the package or available for download from Microsoft:

* A GPU target using DirectX 11 (DirectCompute). This target would directly support integer operations and also allow larger 1-D arrays than the DirectX 9 target.
* An FPGA target.

## Special-Purpose Operations

Plans for Accelerator v2 include several operations that perform special-purpose computations, including:

* A fast-Fourier transform (FFT).
* A general purpose sliding-window filter.
* A random number generator.

## Sets of Data-Parallel Array Objects

Plans for Accelerator v2 allow operations to use sets of data-parallel objects. Programmatically, a set is an array of data-parallel array objects. If a result object’s expression graph contains a set of data-parallel array objects, the target runtime automatically evaluates the associated expression graph once for each object in the set and returns a corresponding array of result objects. If an expression graph contains multiple sets of data-parallel array objects, the target evaluates the graph for the first object in each set, then for the second object, and so on.

If an expression graph contains more than one set of data-parallel array objects, the sets should all have the same number of elements. A target might attempt to handle unequal sets of objects, but it is not guaranteed. Targets typically can do so successfully for only a few simple cases and usually just throw an exception.

To handle sets, Accelerator v2 expression graphs support a hierarchy of data nodes:

1. Constant

2. Data-parallel array object

3. Set of data-parallel array objects

When an expression graph is evaluated, the constants and single data-parallel array objects are coerced up this hierarchy as required.

* Constant values are converted into data-parallel array objects of the appropriate type and shape, with all elements set to the value of the constant.
* If the expression graph contains one or more sets of data-parallel objects, single objects—including those that represent constants—are converted into sets of the same size that are populated with identical objects.

The target runtime determines how to parallelize the evaluation of sets of objects for optimal performance. For example, depending on the particular target, the data objects in the set could be tiled and evaluated on a single processor, scheduled sequentially, or evaluated in parallel on multiple processors. Sets can thus improve performance by providing the target runtime with greater flexibility in how it parallelizes the computation.

## Target API

The currently available targets support the standard evaluation methods, as discussed earlier in this paper, and the DirectX 9 target supports the target memory interface.

Targets can support specialized features, such as a processor-specific fast-Fourier transform, by exposing them as custom operations. These operations are implemented as public methods on the target class, and typically use processor-specific technology for optimal performance. The syntax of a custom operation is usually similar to **ToArray**; it takes a data-parallel object as input and returns an array or bitmap.

An application uses custom operations in much the same way as native Accelerator operations. The primary difference is that custom operations are exposed by a target object instead of Accelerator.dll. Otherwise, Accelerator handles custom operations much like standard operations. They are included in expression graphs, execution is deferred until evaluation, and so on.

**Note:** Custom operations are not yet supported for use by targets.

# Appendix C: Source Code for the C++ Version of StackArrays

This appendix contains the source code for the C++ version of StackArrays.

To run the C++ version of StackArrays

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

2. Open Visual Studio 2008 (or later version) and create a new Win32® console application project.

The project contains two source files. Use the file named for the project to implement the application.

3. Open *ProjectName*.cpp and replace the code with the following example.

4. Open the project’s Properties dialog box.

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

Program Files\Microsoft\Accelerator v2\Include

This is the folder that contains Accelerator.h and DX9Target.h.

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

Accelerator.lib is under the Program Files\Microsoft\Accelerator v2\lib folder. 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**, click **Input** and add Accelerator.lib to the **Additional Dependencies** field.

8. Click **OK** to close the Properties dialog box.

9. Build the application, and copy Accelerator.dll to the project’s 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.

10. Press **CTRL+F5** to run the application.

Listing 4: StackArrays\_CPP

#include "stdafx.h"

#include "Accelerator.h"

#include <D3D9.h>

#include "DX9Target.h"

#include <math.h>

using namespace ParallelArrays; //For all the Accelerator operations

using namespace MicrosoftTargets; //For the DX9Target object

int \_tmain()

{

typedef FloatParallelArray FPA;

int arrayLength = 100;

float\* inputArray1;

float\* inputArray2;

float\* stackedArray;

int randScale = RAND\_MAX \* 10;

DX9Target\* evalTarget = CreateDX9Target();

inputArray1 = new float[arrayLength];

inputArray2 = new float[arrayLength];

stackedArray = new float[arrayLength];

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

{

float angle = (float) i;

inputArray1[i] = (float) (sin(angle/10) + rand()/randScale);

inputArray2[i] = (float) (sin(angle/10) + rand()/randScale);

}

FPA fpInput1 = FPA(inputArray1, arrayLength);

FPA fpInput2 = FPA(inputArray2, arrayLength);

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

FPA fpOutput = ParallelArrays::Divide(fpStacked, 2);

evalTarget->ToArray(fpOutput, stackedArray, arrayLength, ExecutionModeNormal);

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

{

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

}

return 0;

}

# Appendix D: StackMany Source Code

This appendix contains the source code for both versions of StackMany. To build and run either application, see the directions in “Accelerator QuickStart” earlier in this paper.

Listing 5: StackMany

using System;

using Microsoft.ParallelArrays;

using FPA = Microsoft.ParallelArrays.FloatParallelArray;

using PA = Microsoft.ParallelArrays.ParallelArrays;

namespace StackMany1

{

class StackMany

{

static void Main(string[] args)

{

int arrayLength = 100;

int numArrays = 10;

int i, j;

Random ranf = new Random();

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

float[] stackedArray = new float[arrayLength];

FPA fpInput, fpStacked, fpOutput;

DX9Target evalTarget = new DX9Target();

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);

}

}

stackedArray = inputArrays[0];

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

{

fpStacked = new FPA(stackedArray);

fpInput = new FPA(inputArrays[i]);

fpOutput = PA.Add(fpStacked, fpInput);

stackedArray = evalTarget.ToArray1D(fpOutput);

}

fpStacked = new FPA(stackedArray);

fpOutput = PA.Divide(fpStacked, numArrays);

stackedArray = evalTarget.ToArray1D(fpOutput);

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

{

Console.WriteLine(stackedArray[i].ToString());

}

}

}

}

Listing 6: StackMany2

using System;

using Microsoft.ParallelArrays;

using FPA = Microsoft.ParallelArrays.FloatParallelArray;

using PA = Microsoft.ParallelArrays.ParallelArrays;

namespace StackMany2

{

class StackMany2

{

static void Main(string[] args)

{

int arrayLength = 100;

int numArrays = 10;

int i, j;

Random ranf = new Random();

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

float[] stackedArray = new float[arrayLength];

FPA fpInput, fpStacked;

DX9Target evalTarget = new DX9Target(); //Create target object.

for (i = 0; i < numArrays; i++) //Create data arrays

{

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);

}

}

//Stack data-parallel array objects.

fpStacked = new FPA(inputArrays[0]);

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

{

fpInput = new FPA(inputArrays[i]);

fpStacked = PA.Add(fpStacked, fpInput);

}

//Normalize data-parallel array objects and evaluate results.

fpStacked = PA.Divide(fpStacked, numArrays);

stackedArray = evalTarget.ToArray1D(fpStacked);

//Display results.

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

{

Console.WriteLine(stackedArray[i].ToString());

}

}

}

}