Brook+ Specification

This document describes the Brook+ language. Brook+ provides a rapid prototyping tool for developers of high-performance applications to test ideas on stream processor, multi-stream-processors, or multi-core CPU platforms.

1 The Structure of a Brook+ Program

Conventional C code describes a single thread of execution. Although extensions exist at the library level to manipulate threads and processes, the language specification (including the standard library) does not address parallelism.

Brook+ is an extension of C that supports an explicit model of parallelism. As explained below, it is based on a graph consisting of nodes that manipulate data and arcs that indicate the flow of data through the system (see Figure 1 and Figure 2). (Note that this assumes a much more regular and bounded flow of data than is the case for a traditional dataflow machine.)

A node can either restructure data or perform computations, but not both. Nodes that restructure data are called *stream operators*; nodes that perform computations are known as kernels. Both are independent processes that share a state only with that part of the system to which they are explicitly connected. A node starts when the program containing it starts; it executes whenever input data and output buffers are available; it ends when its parent program has completed execution.

An arc, known as a *stream*[1] in Brook+, connects two nodes. It does not provide any storage; instead, it maps the output of one node to the input(s) of one or more other nodes. (Implementations are permitted to introduce intermediate storage for streams, and often do, so long as this storage is transparent to the code.)

Brook+ also provides iterators that linearly interpolate values across a stream. These are like kernels that take no inputs and compute a trivial function of the stream indexes.

The symbols shown in Figure 1 represent the basic building blocks described above.

[1] Streams provide connectivity between processing stages. A stream is a reference to an N-dimensional array of identically-typed primitive elements (a container with a coordinate space); however, it has more restricted access semantics than do conventional arrays. These restrictions permit optimization of both storage requirements and computation locality, providing higher performance for those algorithms that this model can accommodate.

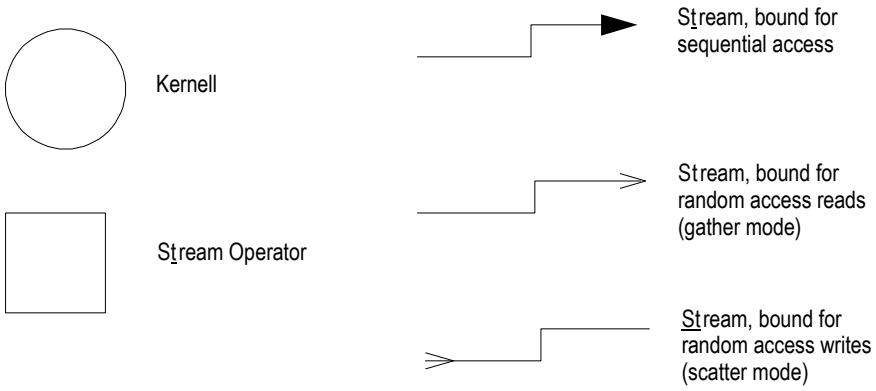

Figure 1 - Symbols for Brook+ Building Blocks

The simple illustration in Figure 2 gives a context for these symbols. It represents a multiply-add operation applied to a 10x20 grid of points.

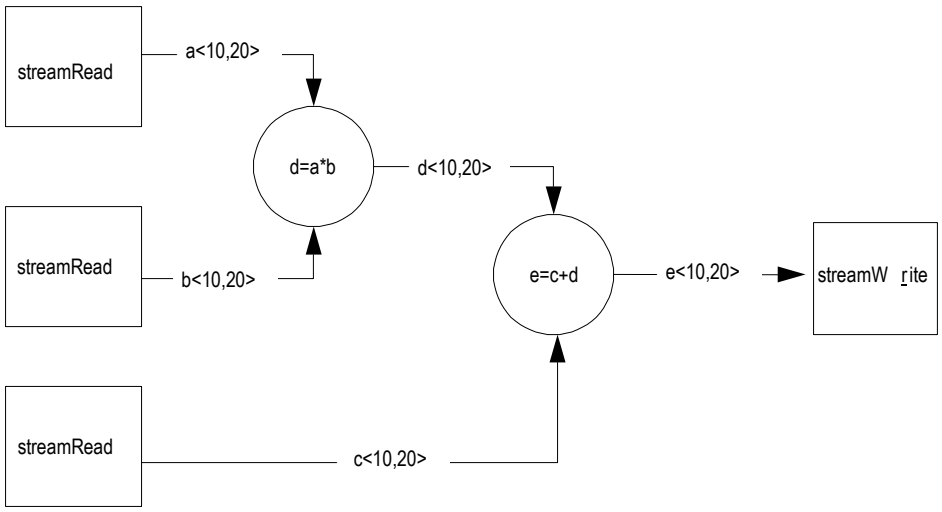

Figure 2 - Simple Streamed Multiply-Add

2 Primitive Data Types

These types can be used as primitive elements.

`int`	32-bit integer, signed by default
`float`	32-bit floating point
`double`	64-bit floating point; this can have a maximum of two elements

These primitive types can be aggregated using `struct` sub-scripting to generate more complex types of stream elements.

For example:

```
struct five_floats
{
   float4 a;
   float b;
};
```

is a valid Brook+ data type.

Brook+ provides built-in short vector types for `float`, `double`, and `int`; this lets code be tuned explicitly for commonly available short-SIMD machines. Here, short vector means 2 to 4 elements long. The names of these types are built from the name of their base type, with the size appended as a suffix (for example: "`float3`", and "`int2`"). These short-vector forms also can be used as primitive elements.

Access to the fields of a short vector type is through structure member syntax, as in standard C code. For example, the float short vectors have the following equivalence:

```
float2 = struct {floatx; floaty}
float3 = struct {floatx; floaty; floatz}
float4 = struct {floatx; floaty; floatz; floatw}
```

When an operator is applied to operands of a short vector type, it is equivalent to applying the operator to each field individually. For example:

```
float2 a, b, c;
c = a + b;
```

is equivalent to:

```
float2 a, b, c;
c.x = a.x + b.x;
c.y = a.y + b.y;
```

Relational Operators on Short Vectors

Relational operators on short vectors in conditional expressions assume an x component as the conditional expression. When using the output of a relational operator as the input to a conditional expression, only the x component of the value is considered. If your application requires full component-wise conditional expressions, you must operate on each component individually.

When you perform an operation on short vectors, the expected behavior is that:

```
float4 a,b
float4 c;
c = a + b;
```

is the same as:

```
c.x = a.x + b.x;
c.y = a.y + b.y;
c.z = a.z + b.z;
c.w = a.w + b.w;
```

However, for relational operators, such as `a < b`, the following code illustrates the difference:

`d = a < b ? a : b` is the same as:

```
bool4 c;
c.x = a.x < b.x;
c.y = a.y < b.y;
c.z = a.z < b.z;
c.w = a.w < b.w;

d.x = c.x ? a.x : b.x;
d.y = c.x ? a.y : b.y;
d.z = c.x ? a.z : b.z;
```

3 Streams and Stream Operators

This section describes the function of streams, the syntax for stream declarations, and how to use stream operators.

3.1 Streams

Streams provide connectivity between processing stages. A stream is a *reference* to an N-dimensional array of identically-typed primitive elements (a container with a coordinate space); however, it has more restricted access semantics than do conventional arrays. These restrictions permit optimization of both storage requirements and computation locality, providing higher performance for those algorithms that this model can accommodate.

Logically, streams do not cause storage to be allocated; however, implementations often allocate large amounts of intermediate storage to contain the data flowing around the system in streams.

As with C arrays, all dimensions but the left-most (slowest changing) must have explicitly specified bounds. The uppermost dimension can be specified implicitly.

3.2 Stream Declarations

The syntax for specifying a stream is similar to other C variable or type declarations, except that angle brackets are used to mark the type/variable as a stream and to delineate the stream dimensions. For example:

`float a<>;` 1D, unspecified length containing `float` elements.

`float b<>[2][3];` 1D, unspecified length containing `float[2][3]` elements.

`int c<100>;` 1D, 100 `int` elements long.

`int d<100,200,300>;` 3D, 100x200x300 `int` elements in size.

`double e<,100>;` 2D, unspecified length but 100 `double` elements wide.

Unspecified lengths are permitted only for declarations that form part of formal parameters[2], all other declarations must specify all sizes explicitly. All dimensions must be integer expressions.

[2] Formal parameters are the names given in the function definition; this is distinct from actual parameters, which are the values passed to the function.

The elements of a stream cannot be accessed from regular C code; they are visible only to kernels and stream operators. (See Section 3.3.1, “I/O Stream Operators”, for more details.)

Streams can contain aggregates of primitive elements, but aggregates of streams are not permitted.

The current implementation supports streams containing up to 2^{23} elements.

3.3 Stream Operators

A stream operator looks like a function call and either:

- remaps a stream, or presents a remapped view of a stream, without changing data at the element level, or
- provides an I/O mechanism between the streaming world of the Brook+ code and the enclosing host environment.

3.3.1 I/O Stream Operators

The following describes copying data to, and from, host (CPU) memory.

Copying Data from Host (CPU) Memory

When reading a stream, it is copied twice: first, from the host (CPU) memory to the PCIe memory, then to the local (stream processor) memory.

The code:

`streamRead(`*`destination_stream`*`, `*`source_array`*`)`

copies the elements of `source_array` to `destination_stream`.

The number of dimensions, size, and element types must match; otherwise, the behavior is undefined.

A `streamRead` operation includes the following order of CAL function calls.

1. `calResMap` maps the memory resource to the stream.
2. `memcopy` copies the data from the data pointer to the stream resource.
3. `calResUnmap` unmaps the memory resource.
4. `calMemCopy` copies the memory to the graphics device. This is required only if the resource was allocated as remote.

Copying Data to Host (CPU) Memory

When writing a stream, it is copied twice: first from local (stream processor) memory to the PCIe memory, then to the main host (CPU) memory. The code:

`streamWrite(`*`source_stream`*`, `*`destination_array`*`)`

copies elements from `source_stream` to `destination_array`.

The number of dimensions, size, and element types must match; otherwise, the behavior is undefined.

A `streamWrite` operation includes the CAL function calls listed above in the following order.

1. `calMemCopy` copies the memory to the graphics device. This is required only if the resource was allocated as remote.
2. `calResMap` maps the memory resource to the stream.
3. `memcopy` copies the data from the data pointer to the stream resource.
4. `calResUnmap` unmaps the memory resource.

3.3.2 Implicit Insertion of Stream Operators

If a kernel is bound to a stream the size of which is different from that specified in the kernel's formal parameter, Brook+ automatically inserts an implicit stream operator that rescales the stream to match. The following examples illustrate this.

The first example is an instance of downscaling from a larger stream to a smaller one.

```
#include <stdio.h>

kernel void copy(float a<>, out float b<>)
{
    b = a;
}

int main(int argc, char **argv)
{
    float src<10>;
    float dst<5>;
    float s[10];
    float d[5];
    int i;

    for (i = 0; i < 10; i++)
    {
        s[i] = (float)i;
    }

    streamRead(src, s);
    copy(src, dst);
    streamWrite(dst, d);

    for (i = 0; i < 10; i++)
    {
        printf("%4.1f ", s[i]);
        if (i < 5)
        {
            printf("%4.1f", d[i]);
        }
        puts("");
    }
}
```

Here, the source stream is twice the size of the destination stream; so the kernel downscales during the copy process by skipping every second element in the input. The result of running this example is:

```
0.0  0.0
1.0  2.0
2.0  4.0
3.0  6.0
4.0  8.0
```

```
5.0
6.0
7.0
8.0
9.0
```

Upscaling is similar:

```
#include <stdio.h>
kernel void copy(float a<>, out float b<>)
{
    b = a;
}
int main(int argc, char **argv)
{
    float src<5>;
    float dst<10>;
    float s[5];
    float d[10];
    int i;
    for (i = 0; i < 5; i++)
    {
        s[i] = (float)i;
    }
    streamRead(src, s);
    copy(src, dst);
    streamWrite(dst, d);
    for (i = 0; i < 10; i++)
    {
        if (i < 5)
        {
            printf("%4.1f ", s[i]);
        }
        else
        {
            printf("     ");
        }
        printf("%4.1f\n", d[i]);
    }
}
```

Here, the situation is reversed, and the kernel upscales the input stream by replicating each element. The result is:

```
 0.0  0.0
 1.0  0.0
 2.0  1.0
 3.0  1.0
 4.0  2.0
      2.0
      3.0
      3.0
      4.0
      4.0
```

4 Kernels

Kernels are the part of the streaming model used to define computation. The most basic form is simply mapped over input data and produces one output item for each input tuple. Subsequent extensions of the basic model provide random-access functionality, variable output counts, and reduction/accumulation operations.

4.1 Kernel Types

There are two kernel types: basic and reduction. The following subsections provide information about each.

4.1.1 The Basic Kernel

The simplest type of kernel takes an element from the same location in each input stream, computes a function of it, then writes it to the corresponding location in the output stream. This is repeated for every element.

```
void kernel mad(float a<>, float b<>, float c, out float d<>)
{
   d = a * b + c;
}
```

All input streams must be of the same size for this operation to be meaningful (however, see Section 3.3.2, “Implicit Insertion of Stream Operators”). When the sizes can be determined at compile-time, implementations are required to check correctness. When the stream sizes cannot be determined at compile-time, provide a compile-time option to enable or disable runtime checking.

The current implementation supports binding 128 inputs and 8 outputs to a single kernel.

4.1.2 Reduction Kernels

Reductions are kernels that decrease the dimensionality of a stream by folding along one axis using an associative and commutative binary operation. The requirement that the operation be associative and commutative means that the result is independent of evaluation order, modulo, any issues due to limited floating point precision.

Brook+ provides two mechanisms for specifying reductions: *reduction variables* and *reduction functions*.

A reduction variable is specified as part of a kernel and operated on using any of the C assignment operators that satisfies the associativity and commutativity requirements; that is: +=, *=, |=, and ^=.

Reduction variables can be any of the primitive types specified above.

For example:

```
void kernel sum(float a<>, reduce float b)
{
   b += a;
}
```

Reduction variables do not necessarily have to be updated for every kernel invocation.

For example:

```
void kernel cond_sum(float a<>, reduce float c)
{
   if (a > 10.0)
   {
      c += a;
   }
}
```

Provide the correct identities (0 for addition, 1 for multiplication, ∞ for max, etc.) as part of the invocation of the reduction.

In addition to the associative assignment operators listed above, the programmer also can specify a *reduction function* that is guaranteed to meet the same requirements. (This is not checked by the compiler.) A reduction function is marked by prefixing the function definition and the reduction variable with the `reduce` keyword[3].

For example:

```
reduce void max_reduce(double a, reduce double b)
{
    if (a > b)
       b = a;
}

reduce void min_reduce(double a, reduce double b)
{
    if (a < b)
       b = a;
}
```

It can be called either as a kernel from the host code, or used as a subkernel by an enclosing kernel (which can itself be a reduction kernel).

Both the input and the output stream of the reduction must be of the same type.

The reduction operator or function must not operate on an expression or function of the input stream element.

Here is an example of a kernel calculating Σ`f(a[i])`, where f(x) is a function of x, and f(x) $\neq$ x.

```
reduce void calc_sum_f(float a<>, reduce float b<>)
{
    b += f(a);
}
```

The result for this kernel is undefined.

4.1.3 Partial Reductions

A partial reduction is possible if the target stream has the same number of dimensions as the source stream. This reduces size but not dimensionality. Each dimension of the source must be: (a) no smaller than the corresponding dimension of the target, and (b) an integer multiple of the corresponding dimension of the target.

For example, assuming a reduction kernel called `sum()`:

```
float s<100,200>;
float t<100>;
sum(s, t);
float u<100,50>;
sum(s, u);
```

Each element of `t` is generated by summing a 1x200 strip from `s`, and each element of `u` is generated by summing a 1x4 strip from `s`.

[3]Currently, prefixing the variable is sufficient to mark the kernel as a reduce kernel.

4.2 Kernel-Specified Communication Patterns

Brook+ is based on a separation of communication and computation, with stream operators defining communication patterns and kernels defining computation. Some users find this too restrictive, so a mechanism has been provided to allow kernels to specify their own communication patterns.

If a stream is bound to a kernel using array brackets rather than stream brackets, the code inside the kernel can access any of the elements of the stream, not just the single element to which the kernel is mapped. This is very similar to a C array operation, except that the index is presented as a float2 (rather than 2 floats in C).

For example:

```
kernel void gather_ex_1(float2 a<>, float b[100][100], out float c<>)
{
    c = b[a];
}
```

Indices can be pulled directly from a stream, or computed as part of the kernel operation:

```
kernel void indexing(float3 a<>, float b[100][100][100], out float c<>)
{
    float3 d = some_function(a);
    c = b[d];
}
```

A stream must be bound write-only or read-only. Read-write binding is not permitted.

Note that specifying communication patterns inside kernels rather than using stream operators can degrade performance.

4.3 Calling Other Code from Kernel Code

Kernels can call other functions defined in the same `.br` file or any files it includes; however, there are restrictions.

- A top-level kernel must have a return type of `void` to be callable from host code. Subkernels can return data of any non-stream type. A subkernel also can be bound to streams propagated from its parent kernel.
- Subkernels are logically expanded inline, so recursion is not permitted.
- Kernels cannot call stream operators.

4.4 Restrictions on Kernel Code

Kernels can use both stream and non-stream parameters as inputs. Generally, only streams can be used as outputs (but see reductions, below).

Within a kernel definition, the following restrictions apply:

- The `goto`, `volatile`, and `static` keywords are prohibited.
- All variables must be of automatic storage class (that is, declared on the stack).
- Pointers are not supported.
- Recursion is not allowed.

- Precise exceptions are not supported.
- Any pointers passed into Brook+ code are required not to alias each other.
- Brook+ functions callable from C code are required to fully specify the sizes of array arguments.
- Storage allocated by Brook+ code can not be accessed by external code except during the lifetime of external functions called from that Brook+ code; and streams are never accessible to non-Brook+ code.

A Brook+ project can be made up of both C/C++ and Brook+ source files, with the Brook+ files having the extension `.br`. Within a Brook+ file, the following restrictions apply:

- Brook+ functions can not call functions declared in C files.
- Preprocessor directives are passed through to the host C++ compiler untouched and uninterpreted.

5 Standard Library Functions and Intrinsics

The following is a listing and description of the kernel intrinsics.

indexof() The indexof operator is applied to a stream and returns a `float` (or `floatN`) type containing the index of the element that the kernel currently being mapped over.

This operator is not valid for reduction or gather streams.

abs(x) Absolute value of `x`.

acos(x) Inverse cosine of `x`.

asin(x) Inverse sine of `x`.

clamp(x,a,b) Clamps the supplied value to be between an upper and lower limit. $a \leq clamp(x) \leq b$.

cos(x) Cosine of `x`.

cross(x,y) Cross product of the two vectors x and y.

dot(x,y) Dot product of the two vectors x and y.

exp(x) e^x

floor(x) $\lfloor x \rfloor$

fmod(x,y) Returns f such that $x = i * y + f$, where i is an integer, f has the same sign as x and $|f| < |y|$.

frac(x) Returns the fractional part of x.

isfinite(x) Returns true if x is finite, false (0) otherwise.

isinf(x) Returns true if x is infinite, false (0) otherwise.

isnan(x) Returns true if x is NaN, false (0) otherwise.

lerp(x,y,a) $(1 - a)x + ay;\ 0 \leq a \leq 1$

log(x) ln (*x*)

max(x,y) Returns the greater of *x* or *y*.

min(x,y) Returns the lesser of *x* or *y*.

normalize(x) Normalizes a vector, returning $\frac{x}{|x|}$.

pow(x,y) x^y

rsqrt(x) $\frac{1}{\sqrt{x}}$

round(x) Rounds *x* to the nearest integer by adding 0.5 and truncating.

sign(x) Returns the sign of *x*, if x is 0 then sign(*x*) is also 0.

sin(x) Sine of *x*.

sqrt $\sqrt{x}$

6 Brook+ Semantic Checker

The following subsections describe built-in data types, type qualifiers, and semantic checks in brcc.

6.1 Built-in Data Types

6.1.1 Supported Built-in Data Types

Scalar types:

`char`	`int`
`uchar` (unsigned char)	`uint` (unsigned int)
`short`	`float`
`ushort` (unsigned short)	`double`

Vector types:

char2
char3
char4
uchar2 (unsigned char2)
uchar3 (unsigned char3)
uchar4 (unsigned char4)
short2
short3
short4
ushort2 (unsigned short2)
ushort3 (unsigned short3)
ushort4 (unsigned short4)
int2
int3
int4
uint2 (unsigned int2)
uint3 (unsigned int3)
uint4 (unsigned int4)
float2
float3
float4
double2

6.1.2 Reserved Built-in Data Types

Scalar types:

long
unsigned long(64 bit)
ulong
long long(128 bit)
unsigned long long
ulong long

Vector types:

longn
ulong
long longn
ulong longn

Where postfix n can be 2, 3, or 4.

6.2 Type Qualifiers

Valid type qualifiers are: const, volatile, restrict, out, shared

.

6.3 Semantic Checks in brcc

6.3.1 Type Qualifier

Qualifier	Non-Kernel Code	Inside Kernel	As kernel Parameter
const	Valid	Valid	Invalid
volatile	Valid	Invalid	Invalid

restrict	Valid	Invalid	Invalid
out	Invalid	Invalid	Valid
shared	Invalid	Valid	Invalid

6.3.2 Storage Classes

Storage Class	Non-Kernel Code	Inside Kernel	As Kernel Parameter
extern	Valid	Invalid	Invalid
static	Valid	Invalid	Invalid
auto	Valid	Valid	Invalid
register	Valid	Valid	Invalid

While `auto` and `register` keywords are accepted by brcc, they currently have no effect on the generated code. brcc `remove`, `auto`, and `register` keywords automatically form declaration statements.

6.3.3. Conversion Rules

Conversion rules are the same as C99.

In the case of vectors of different types, implicit type conversion can be used to promote the smaller type to the larger type.

If the vector types are the same, the vector with the greater size becomes the promoted type

.

```
Ex:  float2 a = float2_(2.0f, 2.0f);
     float2 b = float2_(2.0f, 2.0f);
     float4 d = float4_(2.0f, 2.0f, 2.0f, 2.0f);

     int4 c = int4_(2, 3, 4, 5);

     b = c + a + a; //! c is implicitly converted into float2

     d = d + a; //! a is implicitly converted into float4
```

Expression Type	Can Be Promoted To
char	uchar, short, ushort, int, uint, float, double
uchar	short, ushort, int, uint, float, double
short	ushort, int, uint, float, double
ushort	int, uint, float, double
int	unsigned int, float, double
unsigned int	float, double
float	doubles

By default, brcc enables strong type checking. With strong type checking, no implicit conversions are allowed (both operands must have a same type and number of components).

Use the `-a` flag to disable strong type checking. If the `-a` flag is enabled, warnings are based on the conversion rules explained above.

If strong type checking enabled, all warnings greater than level 1 become errors, and the `-w` and `-x` flags are disabled automatically.

The component count of the casting type must match that of the expression type; otherwise, brcc issues an error. Use the `-a` flag to disable.

6.3.4 Vector Swizzle

The right-hand side of a vector assignment can contain duplicate components. However, the left-hand side of a vector assignment cannot contain duplicate components. One-to-many component assignments are not allowed. For example:

```
float4 a = float4(1.0f, 0.0f, 1.0f, 0.0f)
float4 b = float4(1.0f, 0.0f, 1.0f, 0.0f)
float2 c = float2(1.0f, 0.0f)
a  =  b.xxzw;
a.xx = c; //! Illegal
c. z = a.z //! Illegal

type of c.z is float
type of  a.zz is float2
```

6.3.5 Vector Literals

Here is an example of a constructor vector literal

.

```
float4 a = float4(1.0f, 0.0f, 1.0f, 0.0f)

float2 b = float2(1.0f, 0.0f)
```

brcc now allows vector constructors in any expressions.

Examples:

```
float2 a = float2_(1.f, 1.f);
float2 b = a + float2_(1.f, 1.f);
float x = 1.f;
float2 c;
c = a - float2_(0.f, 0.f);
c = a - float2(x, 0.f);
```

Note the following restriction: an N component vector must have N components specified in the constructor. Each component can be a scalar constant or scalar variable.

Example 1:

```
int n = 1;
int4 imask = int4 (n+2, n-2, n, n);   //! Valid statement
```

Example 2:

```
int2 xx = int2 (1, 1);
int4 imaska = int4 (xx, n, n);   //! This is not allowed
```

When strong type checking is enabled, the values that are given to the constructor must have the same type as the elements of the vector.

Use the `-a` flag to disable strong type checking.

For example:

```
float4 a = float4( 0.0f, 0.0f, 0.0f, 0.0)
```

Note that if strong type checking enabled, brcc issues following error

.

```
sum.br (4): ERROR--1: In Initialization: Mismatched type for element 3: actual type is double but expected type
is float

        Statement: float4 a = { 0.000000f, 0.000000f, 0.000000f, 0.000000 }
```

6.3.6 Semantic Handling of `indexof`, `instance`, `instanceInGroup`, `syncGroup`

Operator	Handling
`indexof`	Always returns float4. Allowed only for streams and scatter. Not allowed on gather arrays or any local variable. Cannot be used in a reduction kernel.
`instance`	Always returns int4. Cannot be used in a reduction kernel.
`instanceInGroup`	Always returns int4. Cannot be used in a reduction kernel. Cannot be used in a pixel kernel.
`syncGroup`	Always returns void type. Cannot be used in a reduction kernel. Cannot be used in a pixel kernel.

Use the `-a` flag to disable strong type checking.

6.3.7 Constant Buffer Support

When constant buffer semantics are in effect, the following conditions apply.

- Size of all gather array dimensions must be specified.
- Total number of elements must be ≤ 4096.
- Maximum number of constant buffers allowed is 10.

For example:

```
kernel void sum6(float b, float a1[5][],  float a[5][5], float a2[][5], float aa<>, out float c<>)
```

In the kernel, sum6, the gather array declarations for a1 and a2 are not valid.

brcc displays the following errors for kernel sum6

:

```
sum.br(65) : ERROR--1: Problem with Array variable declaration: Incomplete array size specification: Specify all
array sizes for cached buffers (e.g. a[5][5]) or no array sizes for Streams (e.g. a[][])

sum.br(65) : ERROR--2: Problem with Array variable declaration: Incomplete array size specification: Specify all
array sizes for cached buffers (e.g. a[5][5]) or no array sizes for Streams (e.g. a[][])
```

Use the `-c` flag to disable constant buffer and allow legacy array declarations.

6.3.8 Semantics of Conditional Expressions

The brcc issues an error if the conditional expression type is vector.

```
Handled for do-while loop, while loop, for loop, ternary operator and if
```

6.3.9 Function Call Semantics

A function definition must exist for any function.

Note that an on-scatter kernel can call a non-scatter kernel; however, a non-scatter kernel cannot call a scatter kernel

. A scatter kernel can call a non-scatter kernel, but not a scatter kernel

.

6.3.10 Function Definition Semantics

Function names must be unique among all function names and variable names in a namespace.

The semantics of kernel parameter declarations are checked against local variable declarations when assignments are made between the two declarations.

The declared return type is compared against the actual return type.

Use the `-a` flag to disable strong type checking.

6.3.11 Array Semantics

brcc supports gather arrays (constant buffers), scatter arrays, and local shared arrays.

For gather arrays, all dimensions must be specified, or none of the dimensions can be specified. Partial dimension specification is not allowed.

Dimension size expression must be an integer constant.

For scatter arrays, you cannot specify any of the array dimensions.

For local shared arrays, you can specify only a 1D array.

Use the `-c` flag to allow gather array declarations, which disables the constant buffer feature.

6.3.12 Operators

Operators	Operates On
Add(-)	Scalars and vectors of char, unsigned char, short, unsigned short, int, unsigned int, float and double.
Subtract(-)	Scalars and vectors of char, unsigned char, short, unsigned short, int, unsigned int, float and double.
Multiply(*)	Scalars and vectors of char, unsigned char, short, unsigned short, int, unsigned int, float and double.
Divide(/)	Scalars and vectors of char, unsigned char, short, unsigned short, int, unsigned int, float and double.
Remainder (%)	Scalars and vectors of char, unsigned char, short, unsigned short, int and unsigned int.
Unary negate(-)	Scalars and vectors of char, unsigned char, short, unsigned short, int, unsigned int, float and double.
Post and pre-increment and decrements(-- and ++)	Scalars and vectors of char, unsigned char, short, unsigned short, int, unsigned int, float and double.
Relation operators (<, <=, >, >=, == and !=)	Scalars and vectors of char, unsigned char, short, unsigned short, int, unsigned int, float and double.
Bitwise operators(&, |, ^, ~, <<, >>)	Scalars and vectors of int and unsigned int.
Logical operators(&& and ||)	Scalars and vectors of char, unsigned char, short, unsigned short, int and unsigned int.
Logical unary operator (!)	Scalars and vectors of char, unsigned char, short, unsigned short, int and unsigned int.
Ternary selection operator (?:)	All valid expressions are allowed.

6.3.13 Index Expression Semantics

Non-C-style and C-style indexing are allowed for all array types.

brcc issues a warning for non-C-style indexing.

6.3.14 Generation of Header File

brcc generates a header file that contains kernel declarations and non-kernel declarations. The name of the generated file is derived from the name of the `.br` file. Header file names can clash with user header filenames.

7 Possible brcc Errors and Warnings

7.1 List of Possible Errors

```
callee unknown

No matched built-in function found

expect int for stream dimension parameter

Scatter Array size must be non-zero positive int value if specified

Scatter Array size must be constant int type if specified

Scatter Array size expression must be simple constant int if specified

Specifying scatter array sizes have no meaning presently. Do not specify sizes

Gather Array size must be non-zero positive int value if specified

Gather Array size must be constant int type if specified

Gather Array size expression must be simple constant int if specified

Incomplete array size specification: Specify all array sizes for cached gather
array (e.g. a[5][5]) or no array sizes for Streams (e.g. a[][])

Local Array not supported yet

Local Array size must be non-zero positive int value

Local Array size must be constant int type

Local Array size expression must be simple constant int

Local Array size expression must be simple constant int

Local Array size expression must be simple constant int

Local Array size expression must be simple constant int

Scatter stream not supported for struct

non-reduce output parameter must be stream type

Stream element type not supported

static or extern not allowed inside kernel
```

```
volatile or Restrict qualifiers not allowed inside kernel

out qualifier allowed only for kernel parameters

reduce qualifier allowed only for kernel parameters

vout deprecated

pointers not allowed inside kernel

Nothing special to use const qualifier for kernel parameters. It is a reserved
usage

Stream can't have an initializer

Iterator not supported

Problem with stream declaration

Problem with stream declaration: dimension == n not supported

Unsupported constant type inside kernel

Initializer is not vector type

Mismatched array dimensions and initializer dimensions

Unsupported variable type

variable has NULL data type

uninitialized const

variable not defined

indexof() operator is not allowed on gather array/local array

indexof()operator allowed only on stream/scatter

Indexof()operator has no meaning inside Non-kernel functions

Instance()operator has no meaning inside Non-kernel functions

conditional resultant type is not a Arithmetic type

String constant in kernel not allowed

Argument n must be lvalue type
```

```
illegal to use components repetition in lvalue expression

unrecognized field/swizzle

indirect call not supported

definition must be available before invoking

callee is not a function

kernel can't call a non-kernel

callee can't call a reduction kernel

incorrect number of parameters expect x actual y

Illegal to use array type at argument n: Expected stream type/variable

Illegal to call scatter kernel from any kernel

Illegal to use array type at argument n: Expected value

Mismatched dimensions of formal and actual arguments for argument n

Mismatched resource type of gather array for argument n

Mismatched element count of gather array for argument n

Local array not allowed to pass as kernel parameter for argument n

Must be gather array type for argument n

Mismatched type for argument n: actual type is “x type” but expected type is “y
type”

Illegal to use array type as unary operand

variable address(&)

pointer dereference(*)

operand is not a Integer type

operand is not a Arithmetic type

double data types can have 1 or 2 components

Non double data types can have 1, 2, 3, or 4 components except void type
```

```
left operand is not a Integer type

right operand is not a Integer type

pointer dereference(->)

Illegal to use array type as left operand

Illegal to use array type as right operand

left operand is not a Arithmetic type

right operand is not a Arithmetic type

Input streams are read only streams

Non-stream/Non Array inputs are not allowed to modify inside kernel

Gather arrays are read only arrays

lvalue is a constant type

Illegal to use array type as casting expression

explicit casting required to have same no of components

Sizeof not allowed in kernel

subscript expression must be scalar type

Variable must be array type

Gather/scatter array dimensions and subscript vector components must match

Index expression size either 1 or equal to array dimensions

Local array variable dimensions and index expression dimensions must be same

Array expression of index expression must be variable type

Switch not supported inside kernel yet

control flow statement not supported

unspecified kernel return type

reduction kernel can only have one input stream and one reduction output
scalar/stream
```

```
Only one scatter output allowed

Both scatter output and stream output not allowed

Variable name Duplicated: variable name

Must return a value

Illegal to use indexof() function in reduction kernel

Illegal to use instance() function in reduction kernel

Illegal to use instanceInGroup() function in reduction kernel

Illegal to use syncGroup() function in reduction kernel

Illegal to use instanceInGroup() for pixel shader kernel

Illegal to use syncGroup() for pixel shader kernel

Illegal to use swizzle on scalar types

Expecting end of comment

Note: comment should end in the same line if it is in the preprocessor directive
line

Expecting token after '#define'

Expecting token after '#undef'

Follow signed integer semantics (only decimal format) for macro value

Undefined preprocessor directive(“TOKEN”)

Unsupported #if syntax

Expecting token after '#ifdef'

Expecting token after '#ifndef'

'#line' is not yet supported

'#include' is not yet supported

Unsupported preprocessor directive syntax

Expecting '#endif' before end of file
```

```
instanceInGroup()operator has no meaning inside Non-kernel functions

syncGroup()operator has no meaning inside Non-kernel functions

one of the multiplication operand must be instanceInGroup().x

offset of Shared array index expression must be a constant integer

offset value(“n”) must be in the range of 0 <= offset < (SharedArraySize /
groupSize)

offset must be a integer constant

stride value must be equal to (SharedArraySize /groupSize)

stride must be a integer constant

Shared array index expression must contain instanceInGroup().x

Shared array allowed only for computer shaders: specify GroupSize Attribute

Only One Shared local array allowed

Shared array type must be 128 bit type

Group size must be <= 1024

Shared array size must be <= 4K 32-bit type

SharedArraySize/GroupSize must be multiples of 4

SharedArraySize/GroupSize must be <= 64

SharedArraySize%GroupSize must be zero

Shared Local Array must be 1D

Shared qualifier valid only for Local Array declared inside kernel

Shared Local Array should not be initialized

conditional expression must be a scalar data type

Output must be a scatter type if you specify GroupSize attribute

More than one GroupSize specified: only one GroupSize allowed

More than 4 components not allowed in GroupSize
```

```
nth dimension of GroupSize must be 1 where n != 1

First dimension of GroupSize must be <= 1024

GroupSize must have values

Unsupported kernel Attribute

Specify reduce qualifier for kernel
```

7.2 List of Possible Warnings

```
language feature not available to dx9 backend

all Parameters of “function name” are converted to float

can't determine on the type of “expression”, expected int

Scatter Array size expression must be simple constant int if specified

Specifying scatter array sizes have no meaning presently. Do not specify sizes

Gather Array size expression must be simple constant int if specified

Incomplete array size specification: Specify all array sizes for cached gather
array (e.g. a[5][5]) or no array sizes for Streams (e.g. a[][])

Uninitialized components

Too many initializers

Mismatched type for element n: actual type is x type but expected type is y type

possible lose of data for element n: conversion from “x type” to “y type”

Initializer is not vector type

Mismatched types: conversion from “x type” to “y type”

Mismatched type: actual type is x type but expected type is y type

Empty array initializer

Mismatched operands: both must have same type and same number of components
```

```
shift count not specified for all components

unused shift count components

implicit conversion from left type to right type: unused left operand components
and possible lose of data

implicit conversion from left type to right type: uninitialized left operand
components and possible lose of data

implicit conversion from left type to right type: unused left operand components

implicit conversion from left type to right type: uninitialized left operand
components

implicit conversion from left type to right type: possible lose of data

implicit conversion from right type to left type: unused right operand components
and possible lose of data

implicit conversion from right type to left type: uninitialized right operand
components and possible lose of data

implicit conversion from right type to left type: unused right operand components

implicit conversion from right type to left type: uninitialized right operand
components

implicit conversion from right type to left type: possible lose of data

shift count not specified for all components

unused shift count components

Mismatched operands: both must have same type and same number of components

Mismatched types: True expression type and false expression type must have same
types and same number of components

Conversion from “x type” to “y type”: possible lose of data and uninitialize
components

Conversion from “x type” to “y type”: possible lose of data and lose of
components

Conversion from “x type” to “y type”: uninitialize components

Conversion from “x type” to “y type”: lose of components

Conversion from “x type” to “y type”: possible lose of data
```

```
explicit casting required to have same no of components

Strongly recommended to use c-style indexing for gather/scatter arrays

possible lose of data for element n: conversion from “x type” to “y type”

kernel required n passes. There could be redundant calculations

“TOKEN”: Macro redefinition

See previous definition of “TOKEN” and latest one considered as macro

Unexpected token following preprocessor directive (#undef “TOKEN”) -- expected a
new line

Unexpected token following preprocessor directive (#ifdef “TOKEN”) -- expected a
new line

Unexpected token following preprocessor directive (#ifndef “TOKEN”) -- expected a
new line

Unexpected token following preprocessor directive (#else) -- expected a new line

Unexpected token following preprocessor directive (#endif) -- expected a new line
```

8 Possible Runtime Errors and Warnings

```
Stream Read: Stream can't read from a NULL pointer

Stream Write: Stream can't write to a NULL pointer

Internal buffer is not initialized

Stream Allocation: Struct streams doesn't support this stream type

Stream Allocation: Stream of this type not supported on the device

Set Property: This is not an interoperable stream

Stream Assign: Destination Internal buffer is not initialized

Stream Assign: Input stream rank and dimension doesn't match with destination

Stream Assign: Uninitialized input stream

Stream Assign: Invalid input stream

Domain Operator: Domain start point can't be NULL
```

```
Domain Operator: Domain end point can't be NULL

Domain Operator: Start point is more than end point, failed to create domain
stream

Domain Operator: Domain dimensions are bigger than original stream, failed to
create domain stream

Domain Operator: Failed to create domain stream

Exec Domain: Internal buffer is not initialized

Exec Domain: Number of threads must be more than 0

Exec Domain: This API is valid only for 1D stream

Exec Domain: Number of threads specified more than stream dimension

Domain Operator: Failed to copy data from parent stream to domain stream

Domain Operator: Failed to copy data from domain stream to parent stream

Kernel Execution: Invalid output stream

Kernel Execution: Invalid Input stream

Kernel Execution: Output stream device is different from current device

Kernel Execution: Input stream device is different from current device

Kernel Execution: Input streams Allocation failed

Kernel Execution: Uninitialized Input streams. The results might be undefined

Kernel Execution: Input stream is same as output stream. Binding kernels read-
write is prohibited

Kernel Execution: No appropriate map technique found

Kernel Execution: Output stream is not allocated

Kernel Execution: output stream rank doesn't match with first output rank. The
results might be undefined

Stream Allocation: Failed to create buffer

Kernel Execution: output stream dimension doesn't match with first output
dimension. The results might be undefined
```

```
Kernel Execution: input stream rank doesn't match with first output rank

Kernel Execution: Failed to create temporary resized input stream

Kernel Execution: Input stream dimension doesn't match with output dimension.
Results might be undefined

Kernel Execution: Input stream dimension doesn't match with output dimension. An
implicit resize is done to match output dimension. In the next release this
feature will be deprecated

Kernel Execution: Setting Execution domain not supported for 3D streams

Assign: Could not flush previous events

Kernel Execution: Reduction operation don't allow output rank > input rank

Kernel Execution: Input dimensions should be integral multiple of output
dimensions. The results might be undefined

Kernel Execution: Unable to create temporary stream required for reduction

Kernel Execution: Failed to execute a reduction pass

Stream Allocation: Rank and dimension of stream doesn't support this stream type

Stream Allocation: Double precision not supported on underlying hardware

Stream Allocation: This dimension not supported on underlying hardware

Stream Read: Could not flush previous events

Failed to map resource

Failed to Initiate DMA

Failed to create host resource

Failed to map tiled resource

Stream Read: Failed to copy data from tiled host resource to stream

Stream Write: Uninitialized stream

Stream Write: Failed to copy data from stream to tiled host resource

Kernel Execution: Failed to create Program

Kernel Execution: Failed to create Constants
```

```
Kernel Execution: Failed to bind constant buffer

Kernel Execution: Error with input streams

Kernel Execution: Error with output streams

Kernel Execution: group size null for compute shader

Kernel Execution: Failed to run kernel

Kernel Execution: Scatter is not supported on underlying hardware

Kernel Execution: Compute Shader is not supported on underlying hardware

Kernel Execution: Failed to create temporary linear stream

Kernel Execution: Failed to copy original data to temporary stream

Kernel Execution: Failed to copy data from temporary stream
```

9 Summary of Command-Line Options Affecting Semantic Checks

Table 1 lists and briefly describes the command-line options for semantic checks.

Table 1 - Semantic Check Command-Line Options

Option	Description
`-a`	Disables strong type checking .
`-c`	Disables constant buffers .
`-wN`	Sets warning level (0, 1, 2, 3) .
`-x`	All warnings are to be treated as errors .

If strong type checking is enabled, all warnings with a level greater than 1 become errors, and the `-w` and `-x` flags are disabled.