Brook+ Programming Guide

This document is for developers using the Brook+ language to develop applications for Stream processors. See *Brook+ Specification* for a complete development guide and language specification.

See the *Brook+_Installation_Notes.pdf* for prerequisites, installation procedures, and Visual Studio syntax highlighting information.

1.1 Runtime Options

Before running Brook+, note the following for Brook+.

- `BRT_RUNTIME` - This environment variable lets you target either the CPU backend (for easy kernel debugging) or the CAL backend (for running on the GPU). If this environment variable is not set, the default is the CAL backend.
  - Set to `cpu` to target the CPU backend.
  - Setting to `cal` to target the CAL backend.
- `BRT_ADAPTER` - This environment variable lets you target a specific GPU in a multi-GPU system. The first GPU is 0, the second GPU is 1, and the n^{th} GPU is N-1. If this environment variable is not set, the default is 0 (the first GPU).
- `BRT_PERMIT_READ_WRITE_ALIASING` - This environment variable lets you bind, at runtime, a stream as both the input stream and output stream. This is not recommended when writing new code. For more information, see the installation notes for Brook+.
- `BRT_LOG_FILE` - This variable let you specify a filename (and its location) that contains internal diagnostic information.

When running Brook+ under Linux, note the following.

- `DISPLAY` - Ensure this is set to `0.0` to point CAL at the local X Windows server. CAL accesses the GPU through the X Windows server on the local machine.
- Ensure your current login session has permission to access the local X Windows server. Do this by logging into the X Windows console locally. If you must access the machine remotely, ensure that your remote session has access rights to the local X Windows server.

1.2 A Sample Application

Brook+ comes in two components: the compiler (`brcc.exe`) and the Brook+ runtime libraries. Building an application consists of:

1. Using the Brook+ compiler to compile the Brook+ source code into a C++ file. This contains the CPU and stream processor code.
2. Compiling the C++ file with the rest of the application and link it with the Brook+ runtime libraries.

The following subsections detail writing, building, executing, debugging, and logging a sample application.

1.2.1 Writing

The following is an example Brook+ source code for `sum.br` that adds two streams and outputs to a third. Brook+ source files normally are given a `.br` extension.

Sum.br

```
#include <stdio.h>

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

int main (int argc, char** argv)
{
   int i, j;
   float a<10, 10>;
   float b<10, 10>;
   float c<10, 10>;

   float input_a[10][10];
   float input_b[10][10];
   float input_c[10][10];
   for (i=0; i<10; i++)
   {
      for (j=0; j<10; j++)
      {
         input_a[i][j] = (float) i;
         input_b[i][j] = (float) j;
      }
   }

   streamRead (a, input_a);
   streamRead (b, input_b);

   sum (a, b, c);

   streamWrite (c, input_c);

   for (i=0; i<10; i++)
   {
      for (j=0; j<10; j++)
      {
         printf ("%6.2f ", input_c[i][j]);
      }
      printf ("\n");
   }

   return 0;
}
```

Brook+ code is very similar to C/C++. Note the following limitations.

First, `brcc` functions like a C compiler; thus, programs must adhere to standard C constructions (for example: variables are declared at the beginning of code blocks).

For more complex applications, carefully partitioning the C code and the Brook+ code into manageable, easily maintainable sections. So, instead of using `main`, a function can be declared there and called from a C/C++ source file.

1.2.1.1 Kernels

From the example on Writing:

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

...

   sum (a, b, c);
```

Kernels are functions that run on the stream processor. The kernel is invoked on every element of the stream. Kernels are executed by calling them, just as in C with the actual parameters.

Kernels are written like C, but with some extensions and limitations (see the *Brook+ Language Specification* for a complete listing). In the following example, *a* and *b* are input streams, and *c* is the output stream. Streams use angle brackets in Brook+. In this situation, the API automatically handles stream addressing.

1.2.1.2 Streams

From the example on Writing:

```
float a<10, 10>;
float b<10, 10>;
float c<10, 10>;
```

Streams are created using angle brackets (rather than square brackets used for arrays in C/C++). The hardware natively supports only 1D arrays up to 8192 elements, and 2D arrays up to 8192x8192 elements, where an element is the stream data type (for example: `float4`). Higher dimensions and larger sizes have limited support through address virtualization at compile time (possibly affecting the performance). For example, a 1D array can be virtualized to 64M (8192x8192) elements. See Building, for enabling address virtualization; also see Section 4.1 of the *Brook+ Language Specification* for more details.

1.2.1.3 Handling Streams

From the example on Writing:

```
streamRead (a, input_a);
streamRead (b, input_b);
...
streamWrite (c, input_c);
```

Streams cannot be accessed directly by the application. Data must be copied between streams and memory using `streamRead()` and `streamWrite()`.

`streamRead (stream *, void *);` Copies data from memory to stream.

`streamWrite (stream *, void *);` Copies data from stream to memory.

1.2.2 Building

Use the following steps to build:

Step 1. Compile with `brcc.exe`, which can be found in
`<BROOKROOT>\sdk\bin\`

```
brcc [-hkrbfilxaec] [-w level] [-D macro] [-n flag] [-w level] [-o prefix]
     [-p shader ] <.br file>
```

Brook+ Parameters

`-h`	Help (print this message).
`-k`	Keep generated IL program (in `<filename.il>`).
`-r`	Disable address virtualization.
`-o <prefix>`	Prefix `prefix` prepended to all output files.
`-p <shader>`	CPU or CAL (can specify multiple).
`-s`	Tokenize into `char` list generated IL program.
`-b`	Turn on bison debugging.
`-f`	Turn on flex debugging.
`-i`	Specify include directory for passing to external preprocessor.
`-l`	Insert #line directives into generated code.
`-w <level>`	Specify warning level: 0, 1, 2, 3; the default is 0.
`-x`	Turn on warnings as errors (valid only with the `-a` parameter).
`-a`	Disable strong type checking.
`-e`	Adds extern C for non-kernel function declarations.
`-c`	Disable cached gather array feature.
`-pp`	Enable the preprocessor.
`-D <name>`	Define a macro.
`-D <name>{=}<int-value>`	Define a macro with an integer value. No spaces are allowed between the macro name and the macro value.
`-n parameter`	Disable the specified parameter. For example, `-n l` disables the line directive information to debug. Currently, only the `-l` parameter is valid with `-n`.

Note that `-x` and `-w` flags are the only ones valid with the `-a` parameter.

Step 2. In the example on Writing, use:

```
brcc.exe -o sum sum.br
```

This compiles the Brook+ file `sum.br` (see Figure1.1 Compiling a Brook+ File and Generating a C++ File) and generates a C++ file, `sum.cpp`, and a `.h` file. Note that the `.cpp` file is output with #line directives; in most cases, this lets you step through the corresponding `.br` file in a debugger.

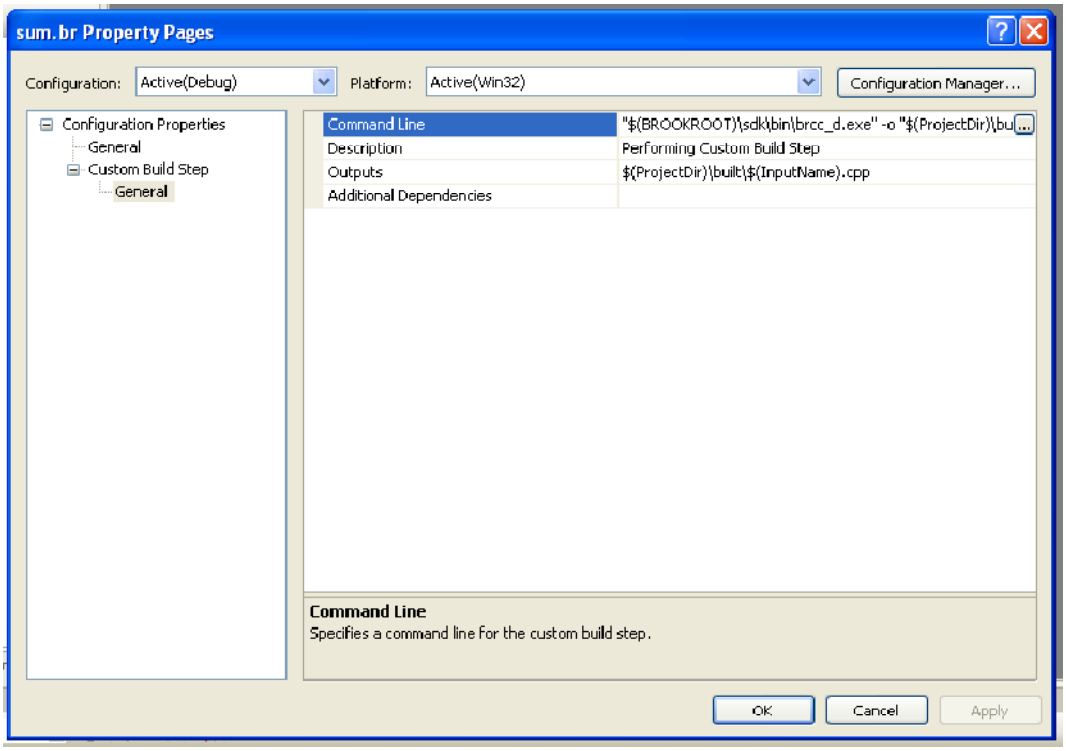

Figure1.1 Compiling a Brook+ File and Generating a C++ File

In Visual Studio, you can add the Brook+ compilation step as a custom build event for the Brook+ file. Right-click on the Brook+ file in the project, and select Properties.

In *Command Line*, add the compiler command. For *Outputs*, add the location of the generated C++ file. You then can add the generated C++ file to the project. Later Brook+ compiles overwrite the existing C++ file.

Brook+ header files are located in `<BROOKROOT>\sdk\include`.

Step 3. Add `brook.lib` to Linker > Input > Additional Dependencies. This library can be found in `<BROOKROOT>\sdk\lib\`.

Step 4. Compile the application with the generated C++ files.

To use a `makefile`, see `<BROOKROOT>\samples\util\build` for examples.

1.2.3 Executing

If the installation was followed correctly and the build was successful, run the executable. If the application does not run, then at least one path has not been set.

1.2.4 Debugging

When debugging an application in Brook+, debugging happens on the generated C++ source, not on the original Brook+ source. For a complete example, see Example of G.

There is no hardware debugging of stream kernels (for example: `__sum_cal_desc`); it is not possible to step through the kernel code. The kernel inputs and outputs can be inspected (before a `streamRead` and after a `streamWrite`). Kernels can be written so that intermediate data can be output to streams and inspected.

Alternatively, kernels can be stepped through and debugged as usual using the CPU emulation mode (for example: `__sum_cpu` and `__sum_cpu_inner`). To enable CPU emulation, create and set the environment variable:

```
BRT_RUNTIME = cpu
```

To return to the CAL backend, either delete the environment variable or set it to:

```
BRT_RUNTIME = cal
```

1.3 Included Samples

The Brook+ folder contains sample applications that can be built using the included makefiles or the included Visual Studio solution file `<BROOKROOT>\samples\samples.sln`.

Release builds of the samples are pre-built and located in:
`<BROOKROOT>\samples\bin\`.

1.3.1 Simple Matrix Multiply Example

This example is a standard matrix multiply. The code presented here is excerpted from the `simple_matmult` example found in the `samples` directory.

```
////////////////////////////////////////////////////////////////////////!
//! C = A * B
//! \param Width The value for which the loop runs over the matrices
//! \param A Input matrix A (MxK)
//! \param B Input matrix B (KxN)
//! \param result Output matrix (MxN)
//!
////////////////////////////////////////////////////////////////////////
kernel void
simple_matmult (float Width, float A[][], float B[][], out float result<>)
{
    // vPos - Position of the output matrix i.e. (x,y)
    float2 vPos = indexof (result).xy;

    // index - coordinates of A & B from where the values are fetched
    float4 index = float4 (vPos.x, 0.0f, 0.0f, vPos.y);
    // step - represents the step by which index is incremented
    float4 step = float4 (0.0f, 1.0f, 1.0f, 0.0f);

    // accumulator - Accumulates the result of intermediate calculation
    // between A & B
    float accumulator = 0.0f;

    // Running a loop which starts from
    // (0,vPos.y) in A and (vPos.x,0) in B
    // and increments the 'y' value of A and the 'x' value of B
    // which basically implies that we're fetching values from
    // the 'vPos.y'th row of A and 'vPox.x'th column of B
    float i0 = Width;
    while (i0 > 0)
    {
        // A[i][k] * B[k][j]
        accumulator += A[index.zw]*B[index.xy];
        index += step;
        i0 = i0 - 1.0f;
    }

    // Writing the result back to the buffer
    result = accumulator;
}
int main (int argc, char** argv)
{
    float A<Height, Width>;
    float B<Width, Height>;
    float C<Height, Height>;
    float* inputA;
    float* inputB;
    float* output;
```

```
    …
    streamRead (A, inputA);
    streamRead (B, inputB);
    …
    simple_matmult ((float)Width, A, B, C);
    …
    streamWrite (C, output);
    …
}
```

Starting at `main`, three streams are created representing the input (`A` and `B`) matrices and the output matrix (streams are used to represent a matrix). Then, three corresponding memory buffers are declared (`inputA`, `inputB`, and `inputC`).

Next, `streamRead()` copies data from `inputA` to stream `A`, and data from `inputB` to stream `B`.

The line `simple_matmult((float)Width, A, B, C);` binds the kernel to the size parameter `Width`, the input streams `A` and `B`, and the output stream `C`; this also triggers execution of the kernel by the stream processor. In a simple matrix multiply operation, the kernel reads in one row vector from one matrix and a column vector from another matrix; it applies a dot product to the two vectors, and writes out the result. In the example above, the kernel is invoked at each data location in the output stream. The kernel:

1. loops over the row of matrix A,
2. loops over the column of matrix B,
3. fetches a value from each matrix, and
4. accumulates the values.

A feature used by this kernel is vector data types (`float2` and `float4`). Brook+ can support data types of up to four elements. Elements can also be accessed in any combination. This is also known as *swizzling*.

There is also a difference between the stream inputs in this kernel compared to those of the earlier sum kernel. Here, the inputs are passed in using square brackets, which means that the input streams are treated as a memory array, and data elements are addressed directly. This is also known as a *gather stream*. An important distinction between kernel code and C code is that gather streams must be accessed using vector types, instead of multiple square brackets. For example, `A[x][y]` is not allowed.

To determine which row/column the kernel must access, the output location to which the kernel is writing must be specified. This is done through the `indexof()` function, which returns an integer (x,y) position of the output domain.

In the `while` loop, column values are read from matrix A and multiplied against values from matrix B. The `accumulator` variable accumulates the resulting values.

Like the earlier sum example, the result is written without bracket operators. Brook+ automatically writes the data out to the correct location; in this case, the location found in `indexof()` of the output stream.

1.3.2 Optimized Matrix Multiply Example

A disadvantage to the above kernel is that the same data is reused by the kernel at separate output locations. For example, at neighboring output locations, the kernel is reusing the same row vector or column vector data. Generally, fetching data from memory is expensive relative to processing data inside the stream processor.

One optimization technique is to perform more computations in the kernel, so that the reads are aggregated. This is the kernel from the optimized matrix multiply sample.

```
kernel void
optimized_matmult (float loopVar0,
        float4 A1[][], float4 A2[][], float4 A3[][], float4 A4[][],
        float4 A5[][], float4 A6[][], float4 A7[][], float4 A8[][],
        float4 B1[][], float4 B2[][], float4 B3[][], float4 B4[][],
        out float4 C1<>, out float4 C2<>, out float4 C3<>,
        out float4 C4<>, out float4 C5<>, out float4 C6<>,
```

```
        out float4 C7<>, out float4 C8<>)
{
    // vPos - Position of the output matrix i.e. (x,y)
    float2 vPos = indexof (C1).xy;

    // Setting four210
    float4 four210 = float4 (4.0f, 2.0f, 1.0f, 0.0f);

    // index - coordinates of A & B from where the values are fetched
    float4 index = float4 (vPos.x, vPos.y, four210.w, four210.w);

    // Declaring and initializing accumulators
    float4 accumulator1 = four210.wwww;
    float4 accumulator2 = four210.wwww;
    float4 accumulator3 = four210.wwww;
    float4 accumulator4 = four210.wwww;
    float4 accumulator5 = four210.wwww;
    float4 accumulator6 = four210.wwww;
    float4 accumulator7 = four210.wwww;
    float4 accumulator8 = four210.wwww;

    float i0 = loopVar0;

    while (i0 > 0.0f)
    {
        // Fetching values from A
        float4 A11 = A1[index.wy];
        float4 A22 = A2[index.wy];
        float4 A33 = A3[index.wy];
        float4 A44 = A4[index.wy];
        float4 A55 = A5[index.wy];
        float4 A66 = A6[index.wy];
        float4 A77 = A7[index.wy];
        float4 A88 = A8[index.wy];

    // Fetching values from B
        float4 B11 = B1[index.xw];
        float4 B22 = B2[index.xw];
        float4 B33 = B3[index.xw];
        float4 B44 = B4[index.xw];
        accumulator1 +=               A11.xxxx * B11.xyzw + A11.yyyy * B22.xyzw +
                        A11.zzzz * B33.xyzw + A11.wwww * B44.xyzw;
        accumulator2 +=               A22.xxxx * B11.xyzw + A22.yyyy * B22.xyzw +
                        A22.zzzz * B33.xyzw + A22.wwww * B44.xyzw;
        accumulator3 +=               A33.xxxx * B11.xyzw + A33.yyyy * B22.xyzw +
                        A33.zzzz * B33.xyzw + A33.wwww * B44.xyzw;
        accumulator4 +=               A44.xxxx * B11.xyzw + A44.yyyy * B22.xyzw +
                        A44.zzzz * B33.xyzw + A44.wwww * B44.xyzw;
        accumulator5 +=               A55.xxxx * B11.xyzw + A55.yyyy * B22.xyzw +
                        A55.zzzz * B33.xyzw + A55.wwww * B44.xyzw;
        accumulator6 +=               A66.xxxx * B11.xyzw + A66.yyyy * B22.xyzw +
                        A66.zzzz * B33.xyzw + A66.wwww * B44.xyzw;
        accumulator7 +=               A77.xxxx * B11.xyzw + A77.yyyy * B22.xyzw +
                        A77.zzzz * B33.xyzw + A77.wwww * B44.xyzw;
        accumulator8 +=               A88.xxxx * B11.xyzw + A88.yyyy * B22.xyzw +
                        A88.zzzz * B33.xyzw + A88.wwww * B44.xyzw;

        index += four210.wwwz;
        // Reducing iterator
        i0 = i0 - 1.0f;
    }

    C1 = accumulator1;
```

```
    C2 = accumulator2;
    C3 = accumulator3;
    C4 = accumulator4;
    C5 = accumulator5;
    C6 = accumulator6;
    C7 = accumulator7;
    C8 = accumulator8;
}
```

This example optimizes the kernel by:

- Using input streams of vector data types. In this case, `float4` is used so that every fetch retrieves four values simultaneously.
- Writing to eight streams simultaneously from the kernel. Using the CAL backend, Brook+ supports up to eight outputs per kernel. Each invocation of this kernel calculates 4×8 = 32 output values. Aggregating the memory fetches per kernel significantly increases the efficiency of the stream processor.
- Separating the two input matrices into multiple slices. This decreases the number of calculations needed to determine the addresses. The same address used to fetch from different inputs representing the slices of the matrices.

Figure1.2 Optimized Matrix Multiplication illustrates the optimized matrix multiplication.

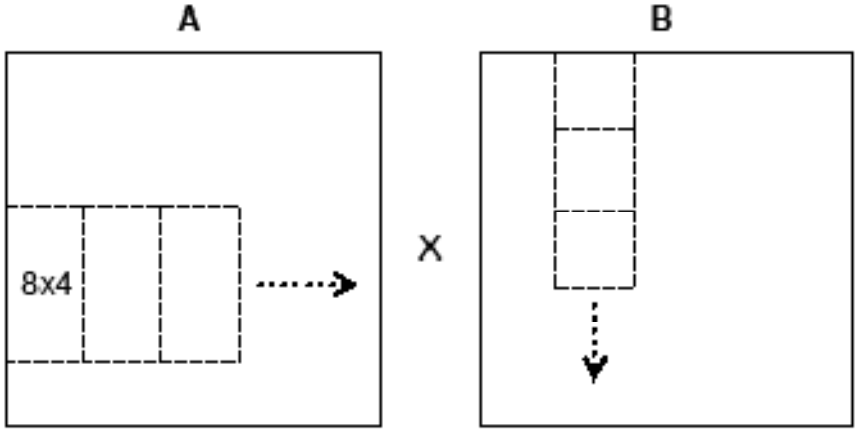

Figure1.2 Optimized Matrix Multiplication

During each iteration of the loop in this kernel implementation, an 8x4 sub-matrix is fetched from matrix A, and a 4x4 sub-matrix is fetched from matrix B. Multiplying these two sub-matrices results in an 8x4 sub-matrix. In the next iteration of the loop, the next 8x4 sub-matrix in the row is fetched from A, and the next 4x4 sub-matrix in the column is fetched from B. These matrices are multiplied and accumulated with the earlier results. The resulting 8x4 matrix is output to a stream.

1.4 Example of Generated C++ Code for `sum.br`

```
////////////////////////////////////////////
// Generated by BRCC v0.1
// BRCC Compiled on: Nov  5 2007 16:24:44
////////////////////////////////////////////

#include <brook/brook.hpp>
#include <stdio.h>


namespace {
  using namespace ::brook::desc;
  static const gpu_kernel_desc __sum_cal_desc = gpu_kernel_desc()
    .technique (gpu_technique_desc()
        .pass(gpu_pass_desc(
            "il_ps_2_0\n"
            "dcl_cb cb0[1]\n"
            "dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)\n"
```

```
            "dcl_input_generic_interp(linear) v0.xy__\n"
            "dcl_resource_id(1)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)\n"
            "dcl_input_generic_interp(linear) v1.xy__\n"
            "sample_resource(0)_sampler(0) r0.x, v0.xy00\n"
            "sample_resource(1)_sampler(1) r1.x, v1.xy00\n"
            "mov r2.x, r0.xxxx\n"
            "mov r3.x, r1.xxxx\n"
            "call 0\n"
            "mov r4.x, r5.xxxx\n"
            "dcl_output_generic o0\n"
            "mov o0, r4.xxxx\n"
            "ret\n"
            "func 0\n"
            "add r6.x, r2.xxxx, r3.xxxx\n"
            "mov r7.x, r6.xxxx\n"
            "mov r5.x, r7.xxxx\n"
            "ret\n"
            "end\n"
            " \n"
            "##!!BRCC\n"
            "##narg:3\n"
            "##s:1:a\n"
            "##s:1:b\n"
            "##o:1:c\n"
            "##workspace:1024\n"
            "##!!multipleOutputInfo:0:1:\n"
            "##!!fullAddressTrans:0:\n"
            "##!!reductionFactor:0:\n"
            "")
            .sampler(1, 0)
            .sampler(2, 0)
            .interpolant(1, kStreamInterpolant_Position)
            .interpolant(2, kStreamInterpolant_Position)
            .output(3, 0)
      )
    );
  static const void* __sum_cal = &__sum_cal_desc;
}

static const char *__sum_ps30= NULL;
void  __sum_cpu_inner(const __BrtFloat1  &a,
                      const __BrtFloat1  &b,
                      __BrtFloat1  &c)
{
  c = a + b;
}
void  __sum_cpu(::brook::Kernel *__k, const std::vector<void *>&args)
{
  ::brook::StreamInterface *arg_a = (::brook::StreamInterface *) args[0];
  ::brook::StreamInterface *arg_b = (::brook::StreamInterface *) args[1];
  ::brook::StreamInterface *arg_c = (::brook::StreamInterface *) args[2];

  do {
    Addressable <__BrtFloat1  > __out_arg_c((__BrtFloat1 *) __k->FetchElem(arg_c));

    __sum_cpu_inner (Addressable <__BrtFloat1 >((__BrtFloat1 *) __k->FetchElem(arg_a)),
        Addressable <__BrtFloat1 >((__BrtFloat1 *) __k->FetchElem(arg_b)), __out_arg_c);

    *reinterpret_cast<__BrtFloat1 *>(__out_arg_c.address) =
              __out_arg_c.castToArg(*reinterpret_cast<__BrtFloat1 *>
    (__out_arg_c.address));
  } while (__k->Continue());
}

void  sum (     ::brook::stream a,
            ::brook::stream b,
            ::brook::stream c) {
  static const void *__sum_fp[] = {
     "ps30", __sum_ps30,
     "cal", __sum_cal,
     "cpu", (void *) __sum_cpu,
     NULL, NULL };
  static ::brook::kernel  __k(__sum_fp);
```

```
  __k->PushStream(a);
  __k->PushStream(b);
  __k->PushOutput(c);
  __k->Map();

}


int  main(int  argc, char  **argv)
{
  int  i;
  int  j;
  ::brook::stream a(::brook::getStreamType(( float  *)0), 10 , 10,-1);
  ::brook::stream b(::brook::getStreamType(( float  *)0), 10 , 10,-1);
  ::brook::stream c(::brook::getStreamType(( float  *)0), 10 , 10,-1);
  float  input_a[10][10];
  float  input_b[10][10];
  float  input_c[10][10];

  for (i = 0; i < 10; i++)
  {
    for (j = 0; j < 10; j++)
    {
      input_a[i][j] = (float ) (i);
      input_b[i][j] = (float ) (j);
    }

  }

  streamRead(a, input_a);
  streamRead(b, input_b);
  sum(a, b, c);
  streamWrite(c, input_c);
  for (i = 0; i < 10; i++)
  {
    for (j = 0; j < 10; j++)
    {
      printf("%6.2f ", input_c[i][j]);
    }

    printf("\n");
  }

  return 0;
}
```

1.5 Building Brook+

Both the release and debug builds of the Brook+ compiler and runtime libraries come pre-built; however, they also can be built using the provided source.

The path to the pre-built SDK (binary, library, and headers) is:

```
<BROOKROOT>\sdk\
```

On Windows systems, Brook+ can be built either from the command line or inside Visual Studio. Either way requires a full install of Cygwin (www.cygwin.com).

1.5.1 Visual Studio

You can build the brcc and the Brook+ runtime using the included Visual Studio solution file, which is located at:

```
<BROOKROOT>\platform\brook.sln
```

The configuration for getting the Debug or the Release executable is available through the Configuration pull-down menu.

The default output directories of builds using Visual Studio are:

```
brcc.exe: <BROOKROOT>\platform\brcc\bin\xp_x86_32
brook.lib: <BROOKROOT>\platform\runtime\lib\xp_x86_32
```

Files in the SDK tree are not replaced with the new builds. If `make` is installed, in `<BROOKROOT>\platform`:

- run `make updatesdk` to copy the debug to the SDK tree,
- or run `make udpatesdk RELEASE=1` to copy the release builds to the SDK tree.

1.5.2 Command Line

The Brook+ tools can be built from the command line or through a Cygwin shell.

1. The Visual Studio compiler (`cl.exe`) and linker (`link.exe`) must be in the path. Default location is:

   `C:\Program Files\Microsoft Visual Studio 8\VC\bin`

   Note that in the path, the Visual Studio `link.exe` must come before the Cygwin `link.exe`.
2. Run *make* at `<BROOKROOT>\platform\` for a debug build and run *make RELEASE=1* for a release build.

Unlike the Visual Studio builds, the SDK tree is rebuilt and overwritten with the new Brook+ builds.

To clean the build, use `make clean` for debug builds and `make clean RELEASE=1` for release builds.

1.6 The Brook+ Runtime API

The current version of Brook+ features a completely rewritten runtime engine. In addition to improvements in performance and stability, there is a new C++ API available for developers looking for a lower-level and more flexible way to access the GPU.

1.6.1 Differences Between the C++ API and the Previous Programming Model

Differences between this and the previous programming model include:

- dynamic stream management
- error handling
- execution domain control
- explicit control over asynchronous APIs
- memory pinning
- multi-GPU support
- DX interoperability
- compatibility with C++ code

The following subsections discuss these differences.

1.6.1.1 Dynamic Stream Management

Brook, BrookGPU, and the legacy version of Brook+ use a statically allocated stream graph and prohibit streams that are bound for simultaneous read and write. At the C++ API level, there are no such restrictions: streams are proxies for GPU memory and can be dynamically allocated and passed between functions like any other C++ object.

1.6.1.2 Error Handling

Errors are now trapped by the runtime and communicated back to the client. As GPU-side errors can be asynchronous relative to host-side control flow, the error is not passed directly back to the host; instead, it is associated with a stream

and propagated through the stream graph. The application checks the final output stream to find out if an error occurred in the process.

1.6.1.3 Execution Domain Control

When using a scatter stream as an output, it is not useful to enforce a simple one-to-one mapping between the layout of the output stream and the layout of the execution domain (the “virtual SIMD array” that runs the kernels).

We now provide an extensible and optional mechanism to supply additional parameters to a kernel invocation.

1.6.1.4 Explicit Control Over Asynchronous APIs

Brook+ now lets you explicitly request that certain stream operations are done asynchronously. An API is provided to check on the status of the asynchronous request. This results in better simultaneous use of the CPU and the GPU, resulting in higher overall system efficiency.

See section Explicit Control Over Asynchronous APIs, for more information on how to explicitly request asynchronous stream operations.

1.6.1.5 Memory Pinning

Memory pinning takes advantage of a system feature that allows CPU-to-GPU and GPU-to-CPU memory transfers directly from, and to, user memory. Normally, data transfers involve a copy into a special memory space on the CPU side before being copied either to the GPU or to user memory. The use of memory pinning improves data transfer performance when possible.

There are certain restrictions on memory pinning usage of which developers must be aware. See Memory Pinning, for more information on how to request memory pinning for your stream operations.

1.6.1.6 Multi-GPU Support

More systems are configured with two or more GPUs. Brook+ lets developers use a single Brook+ program to take advantage of all compatible GPUs in a system. Routines are provided that let the user discover and select the devices to execute on.

1.14 See Multi-GPU Support

, for more information on how to take advantage of multiple GPUs in your Brook+ program.

1.6.1.7 DX Interoperability

DX interoperability lets Brook+ programmers easily and efficiently display the results of their computations on the screen using a familiar graphics API. This is particularly important in image and video processing applications. Interoperability allows the programmer to avoid having to copy the generated data back to the CPU before rendering, improving overall application performance.

See DX Interoperability, for more information on how to use DX interoperability to render your generated data from Brook+.

1.6.1.8 Compatibility With C++ Code

Kernel code is still restricted to a subset of C, but moving all other code outside the `.br` file means that developers can write their application in C++.

1.6.2 Choosing a Programming Model

We recommend that new projects use the C++ API rather than the legacy model. The only reasons for using the legacy Brook interface are:

- for compatibility with other Brook implementations, or
- to benefit from potential compiler improvements, or
- the project is very small and/or simple.

Example

Consider this code fragment:

```
kernel sum(double a<>, double b<>, out double c<>)
{
    c = a + b;
}

void vector_add(double *in_a, double *in_b, double *out, unsigned int length)
{
    double s1<length>, s2<length>, s3<length>;

    streamRead(s1, in_a);
    streamRead(s2, in_b);

    sum (s1, s2, s3);

    streamWrite(s3, out);
}
```

Several limitations of the legacy model are exposed:

- Not all hardware has support for doubles; but there is no way of handling this. See page D-1 for a list of devices that support this feature.
- If there is not enough memory to allocate any of the streams, the program terminates.
- Data can only be passed around by host-side code in host-side memory, potentially requiring multiple extra copies.

Using the new API, this code looks like:

```
kernel sum(double a<>, double b<>, out double c<>)
{
    c = a + b;
}

Stream<double> *vector_add(double *in_a, double *in_b, unsigned int length)
{
    Stream<double> s1(1, length);
    Stream<double> s2(1, length);
    Stream<double> *s3 = new Stream<double>(1, length);

    s1.read(in_a);
    s2.read(in_b);

    sum(s1, s2, s3);

    if (s3->error())
        {
        delete s3;
        return NULL;
        }
    return s3;
}
```

Note that kernel definitions have not changed from the 1.2 format. All the differences are on the host side. Looking at the changes line by line, we have:

- The `vector_add` function now returns a pointer to a stream.
- The three streams (s1, s2, s3) are allocated as C++ objects using a templated constructor.

- `streamRead()` and `streamWrite()` are now methods of the Stream<> class.
- Stream objects now track errors instead of aborting. (For more details on the error handling mechanism, see Public Methods).
- Streams can be passed around by host-side code, removing the need for redundant copies.

1.7 Stream Management (`Stream.h`)

The classes and functions in this file provide a mechanism for creating and managing streams. At this level of abstraction, a stream effectively is a proxy object for a remote array and some error-tracking information. (Other stream semantics are part of the Brook+ language definition and are not enforced by the runtime.)

Backend-specific details are not visible at this level.

1.7.1 Public Methods

The Stream class exposes the following public methods.

```
Stream::Stream(unsigned short rank, unsigned int* dimensions)
Stream::Stream(unsigned short rank, unsigned int* dimensions,
               const char* type)
```

where:

`rank`	Number of dimensions in the stream.[1]
`dimensions`	Upper bound of each dimension. (Array indices run from 0 to dimensions[n]-1 as in conventional C code).
`type`	[Optional] Interoperability data type. Used in creating a stream object for exchanging data with other programming APIs such as DX. See DX Interoperabilityfor more information on how to use this field for DX interoperability.

Use the first constructor when the application code creates a stream. The underlying representation is determined by the backend being used and is transparent to the client.

Use the second constructor when the application code creates a stream for use with DX interoperability. Additional properties must be set when using a stream created in this way.

If creation fails, the stream error state is set to `BR_ERROR_DECLARATION`.

Examples:

```
unsigned int n = 10000; // 1D double
Stream<double> s(1, &n);

unsigned int dims[2] = {1024, 1024}; // 2D float
Stream<float> *s = new Stream<float>(2, dims);

explicit Stream::Stream(StreamImpl* streamImpl)
```

where:

`streamImpl` is the pointer to underlying stream implementation.

[1] This is similar to, but not exactly the same as, “rank” in the mathematical sense.

This is intended only for internal API use. It wraps a backend-specific stream implementation in a generic Stream container.

`void Stream::read(const void* ptr, const char* flags = NULL)`

This copies data from a host-side pointer to the memory associated with a stream. It is equivalent to `streamRead()` in the legacy API. For the CAL backend, this includes a copy over the PCI Express bus; however, transfer speed has been improved greatly compared to the legacy implementation.

Note that the runtime does not check that `ptr` points to a sufficiently large area of memory. This is the programmer's responsibility.

The `flags` parameter controls the behavior of the stream read when requesting asynchronous operations and memory pinning. See the respective sections in this document to learn more about what flags to set when using each of these features. Multiple flags can be specified by separating each flag with a space.

`void Stream::write(void* ptr, const char* flags = NULL) const`

This copies data from the memory associated with a stream to a host-side pointer. This is equivalent to `streamWrite()` in the legacy API.

This is a synchronous call and blocks any return to the caller until all data has been written to the host. (For the CAL backend, this includes a copy over the PCI Express bus; however, transfer speed has been improved greatly compared to the legacy implementation.)

Note that the runtime does not check that `ptr` points to a sufficiently large area of memory, this is the user's responsibility.

The `flags` parameter controls the behavior of the stream write when requesting asynchronous operations and memory pinning. See the respective sections in this document to learn more about what flags to set when using each of those features. Multiple flags can be specified by separating each flag with a space.

`void Stream::assign(Stream<T> source)`

This copies data from a source stream to the current stream. The source stream can be on the same device as the current stream or on a different device. The source stream be of the same type `T` as the current stream.

`Stream<T> Stream::domain(unsigned int* start, unsigned int* end) const`

This extracts a sub-region of interest from the `Stream` lying between the start and end positions in the stream. The routine returns another stream that corresponds to the selected region.

The new stream is treated as a sub-region within the original stream; however, unlike the legacy API, modifications to the child are not guaranteed to be immediately reflected in the parent. Instead, changes can be propagated at any point between them occurring in the child and the child's destructor being called. (A change made in the child can become visible in the parent at any point between it first happening and the child ceasing to exist.)

`void Stream::setProperty(const char* name, void* value)`

This sets the specific properties of a stream. Setting properties is not required for normal stream operations. Currently, this method is used when interoperating with other programming APIs, such as DX.

`bool Stream::isSync()`

This returns the completion status of outstanding asynchronous operations on this stream.

`bool Stream::finish()`

This waits for all asynchronous operations on this stream to complete.

`BRerror Stream::error()`

This checks if an error occurred during processing of this stream or any of the streams from which it was computed. Returns an error code (`enum`) for first error that occurred, or `BR_NO_ERROR` if no error occurred.

The error state is cleared when the `error()` routine is called.

Error Codes:

```
enum BRerror
{
```

```
    BR_NO_ERROR = 0,            // No error. All's well
    BR_ERROR_DECLARATION,       // Error in Stream Declaration
    BR_ERROR_READ,              // Error during Stream::read
    BR_ERROR_WRITE,             // Error during Stream::write
    BR_ERROR_KERNEL,            // Error during Kernel Invocation
    BR_ERROR_DOMAIN,            // Error in domain operator
    BR_ERROR_INVALID_PARAMATER, // An invalid parameter was passed
    BR_ERROR_NOT_SUPPORTED      // Feature not supported in brook+
                                //or in the underlying hardware
};

const char* Stream::errorLog();
```

Returns NULL-terminated `char` string with log messages. Unlike the `error()` call, which records only the first error that occurred, `errorLog()` accumulates a list of all errors from the first onward.

Any error that occurs on a stream is propagated inside the Brook+ data flow pipeline to tag other streams as being in an error state. For example, if an input stream used in a kernel invocation contains an error, the subsequent output stream also is flagged as erroneous. As host and runtime code are potentially asynchronous, it is not practical to check for errors after every stream-related routine invocation. Whenever an error occurs, the stream class appends that error to an internal error log. The `Stream::errorLog()` lets you read this error log in the form of a C string.

Example

Here is an example that illustrates the usage of this interface.

```
int copy(const void *inputPtr, void *outputPtr, unsigned int dims[2])
{
    Stream<float> X(2, dims), Y(2, dims);

    // Initialize X
    X.read(inputPtr);

    // Invoke the kernel
    copy(X, Y);

    // Copy Y back
    Y.write(outputPtr);
    if(Y.error())
    {
        std::cerr >> "Error in Stream Y" >> Y.errorLog() >> std::endl;
        return -1;
    }
}
```

`operator Stream::StreamImpl*() const`

Intended only for internal API use. Returns a pointer to the backend-specific stream implementation inside a generic stream container.

`Stream::~Stream()`

Destroys the proxy object. Actual deallocation of the underlying resources might not happen immediately as some backends use a lazy allocation strategy to improve performance.

1.7.2 Public Data

None.

1.7.3 Compatibility

A preprocessor macro, `USE_OLD_API`, is defined at the top of this file and used to enable/disable support for certain legacy API functions.

When this flag is enabled, the following additional functions and methods are available:

```
    Stream<T> domain(int start, int end) const;
    Stream<T> domain(int2 start, int2 end) const;
    Stream<T> domain(int3 start, int3 end) const;
    Stream<T> domain(int4 start, int4 end) const;
    Stream<T> execDomain(int numThreads) const;

template<class T>
void streamRead(brook::Stream<T> stream, void* ptr);

template<class T>
void streamWrite(brook::Stream<T> stream, void* ptr);
```

These work as they did in the legacy Brook+ API.

1.7.4 Backend Performance

The current implementation supports two backends: CPU emulation and GPU via CAL. The 1.3_beta CAL backend offers significantly better performance compared to both the CPU backend and the 1.2_beta CAL implementation.

1.8 Kernel Management

Invoking kernels in Brook+ is usually as simple as calling a C function with the same name and arguments as the kernel defined by the application in the `.br` file. Generally, the runtime handles all the mapping and device management transparently, but in some situations the user might require direct control of backend-specific features. To enable this, we provide a lower-level kernel interface API, as described below.

For each kernel, brcc generates an overloaded C++ operator for the *KernelInterface* that provides a mechanism for overriding some or all of the defaults.

The current CAL implementation lets the user override the domain of execution of the kernel launch. This is extremely useful for cases where the execution domain is not uniquely defined by the kernel parameters (for example: when using scatter outputs).

```
class KernelInterface
{
public:

  // Constructor and Destructor

  KernelInterface();
  ~KernelInterface();

  // Methods to control domain of execution

  void domainOffset(uint4 offset);
  void domainSize(uint4 size);
};
```

Example:

The following kernel performs random access writes to a scatter stream by using indices from another stream index.

```
kernel void
scatter(float index<>, float a<>, out float b[])
```

```
{
    b[index] = a;
}
```

To set the domain of execution parameters and launch the kernel:

```
scatter.domainOffset(offset);
scatter.domainSize(size);
scatter(index, a, b);
```

1.9 Scatter/Gather Interface Changes

In addition to the KernelInterface feature described above, the new API provides an improved alternative to the `indexof()` intrinsic, `instance()`.

Unlike `indexof()`, `instance()` always integers not floats, and is always a four-element vector (zero-padded where appropriate). This makes for much simpler code, as shown below.

```
kernel void
simple_matmult_indexof(float Width, float A[][], float B[][], out float C<>)
{
   float2 pos = indexof(C).xy;
   float4 ind = float4(pos.x, .0f, .0f, pos.y);
   float4 step = float4(.0f, 1.0f, 1.0f, 0.0f);
   float prod = 0.0f, i0 = 0.0f;

   for(i0 = 0.0f; i0 < Width; i0 += 1.0f)
   {
        prod += A[ind.zw] * B[ind.xy];
        ind += step;
    }

    // Writing the result back to the buffer
    C = prod;
}

kernel void
simple_matmult_instance(uint Width, float A[][], float B[][], out float C<>)
{
   uint4 pos = instance();

   float prod = 0.0f;
   uint i0 = 0;

   for(i0 = 0; i0 < Width; i0++)
   {
        prod += A[pos.y][i0] * B[i0][pos.x];
        index += step;
   }

    // Writing the result back to the buffer
    C = prod;
}
```

1.10 Converting Code to Use the New C++ API

The C++ API is recommended for all future code because it offers greater flexibility and access to more features than the legacy Brook+ API. The following example explains how to convert existing legacy code to use the new API. This example translates the Binary Search sample application supplied in the SDK.

Table 1.1 Ke provides a side-by-side comparison of the kernel code as used in the legacy interface and the new API. For clarity, large sections of code are omitted.

Table 1.1 Kernel Code Comparison: Legacy vs New API

Legacy	New API

kernel void binary_search(float searchValue<>, float array[], out float index<>, float arraySize, float lgWidth) {
 	
  float i;
 	
  float numIter = lgWidth;
  float stride;
  float compareValue, dir;
 	
  float idx = stride = floor((arraySize * 0.5f) + 0.5f);
 	
  index = 0.0f;
 	
  for (i = 0.0f; i < (numIter); i += 1.0f) {
  stride = floor((stride * 0.5f) + 0.5f);
  compareValue = array[idx];
  dir = (searchValue <= compareValue) ? -1.0f : 1.0f;
  idx = idx + dir * stride;
  }
 	
  // last iteration has stride fixed at 1
  compareValue = array[idx];
  idx = idx + ((searchValue <= compareValue) ? -1.0f : 1.0f);
 	
  // last pass check
  compareValue = array[idx];
  idx = idx + ((searchValue <= compareValue) ? 0.0f : 1.0f);
  if (idx < 0.0f)
  {
  idx = 0.0f;
  }
 	
  // if we've found the value, write the array index into the output, otherwise, write -1
  compareValue = array[idx];
  idx = (searchValue == compareValue) ? idx : -1.0f;
 	
  index = idx;
}
	kernel void binary_search(float searchValue<>, float array[], out float index<>, float arraySize, int lgWidth)
	{
	  float stride;
	  float compareValue, dir;
	
	  float idx = stride = floor((arraySize * 0.5f) + 0.5f);
	
	  int i;
	  for (i = 0; i < lgWidth; ++i)
	  {
	  stride = floor((stride * 0.5f) + 0.5f);
	  compareValue = array[idx];
	  dir = (searchValue <= compareValue) ? -1.0f : 1.0f;
	  idx = idx + dir * stride;
	  }
	
	  compareValue = array[idx];
	  idx = idx + ((searchValue <= compareValue) ? -1.0f : 1.0f);
	
	  // last pass check
	  compareValue = array[idx];
	  idx = idx + ((searchValue <= compareValue) ? 0.0f : 1.0f);
	  if (idx < 0.0f)
	  {
	  idx = 0.0f;
	  }
	
	  // if we've found the value, write the array index into the output, otherwise, write -1
	  compareValue = array[idx];
	  idx = (searchValue == compareValue) ? idx : -1.0f;
	
	  index = idx;
	}

In summary, very little has changed between the two versions. There have been minor cleanups, but kernel code remains essentially unchanged.

Table 1.2 Ho provides a side-by-side comparison of the host code as used in the legacy interface and the new API.

Table 1.2 Host Code Comparison: Legacy vs New API

Legacy	New API

```
int main(int argc, char** argv)
{
    unsigned int     i   = 0;
    unsigned int lgWidth = 0;
    float*  array        = NULL;
    float*  searchValues = NULL;
    float*  indices[2]   = { NULL };
    unsigned int Length, Searches;

    {
        float   searchValueStream<Searches>;
        float   indicesStream<Searches>;
        float   arrayStream<Length>;

        // Record GPU Total Time
        Start(0);

        for (i = 0; i < cmd.Iterations; ++i)
        {
            // Copy searchable data and search
keys to streams
            streamRead(arrayStream, array);
            streamRead(searchValueStream,
searchValues);

            // Execute parallel binary search
            binary_search(searchValueStream,
arrayStream,

            // Copy results from stream
            streamWrite(indicesStream,
indices[0]);
        }
    }
}
```

```
#include "brookgenfiles/binary_search.h"

int BinarySearch::run()
{
    unsigned int retVal = 0;

    // Brook code block
    {
        unsigned int arrayDim[] = {_length};
        unsigned int searchDim[] = {_width};

        ::brook::Stream<float> searchValueStream(1,
searchDim);
        ::brook::Stream<float> indicesStream(1,
searchDim);
        ::brook::Stream<float> arrayStream(1,
arrayDim);

        for (unsigned int i = 0; i < info-
>Iterations; ++i)
        {
            // Copy searchable data and search keys
to streams
            arrayStream.read(_array);
            searchValueStream.read(_searchValues);

            // Execute parallel binary search
            binary_search(searchValueStream,
arrayStream,
                          indicesStream,
(float)(_length), _lgWidth);

            // Copy results from stream
            indicesStream.write(_indices[0]);

            //Handle errors if occured
            if(indicesStream.error())
            {
                std::cout << "Error occured" <<
std::endl;
                std::cout <<
indicesStream.errorLog() << std::endl;
                retVal = -1;
            }
        }

        timer->Stop();
    }
    return retVal;
}
```

As can be seen from Table 1.2 Ho, the host-side code has changed considerably.

At a project structure level, host code and kernel code now are in different files. The kernel code lives in `.br` files as before, but the host-side code is in regular C++ source files. The Brook+ compiler, brcc, compiles Brook+ source to a header file (here `brookgenfiles/binary_search.h`) containing all the internal definitions and bindings required by the C++ runtime. This file must be included by the host-side source file.

Also note that the stream definitions have changed. Legacy-mode stream definitions are static and use extended syntax. C++ API definitions are conventional object instantiations, meaning that they can have their addresses taken and passed between functions, as with any other object. Reading from, and writing to, streams now is a method of the Stream class.

Finally, streams now maintain an error status that can be queried by the host to check if a computation (or string of computations, since errors propagate between streams) completed correctly.

1.11 8-/16-Bit Integer Support

brcc now supports the following 8- and 16-bit data types.

Type	Description
char	signed char, 8-bit
uchar	unsigned char, 8-bit
short	signed short, 16-bit
ushort	unsigned short, 16-bit

These data types internally use integer instructions. brcc imposes the following restrictions on these data types:

- constant literals must be type cast explicitly. For example, `short var = 5; // Error`
- Whereas,
- `short var = (short)5 //correct`
- Transcendental operations are not supported natively for these data types. To use transcendental operations on these data types, the application first must type cast it to a float.

The following program demonstrates how to use these new data types.

```
// Kernel
kernel
void sum(char i0<>, char i1<>, out char o0<>)
{
    char var = (char)5;
    o0 = i0 + i1 + var;
}

// Utility method to fill input buffes with value
void
fillBuffer(char *in, char val, unsigned int dimension)
{
    unsigned int i = 0;
    for(i = 0; i < dimension; i++)
    {
        in[i] = val;
    }
}

// Main
int
main(void)
{
    // Input buffers
    char* input0 = NULL;
    char* input1 = NULL;

    // Output buffers
    char* output0 = NULL;

    unsigned int width = 64, height = 64;
```

```
    // Allocate memory for inputs
    input0 = (char*) malloc(sizeof(char)* width * height);
    input1 = (char*) malloc(sizeof(char)* width * height);

    // Allocate memory for outputs
    output0 = (char*)malloc(sizeof(char)* width * height);

    // Fill the input buffers with values
    fillBuffer(input0, 60, width * height);
    fillBuffer(input1, 5, width * height);

    // Brook+ code
    {
        // Declare input/output streams
        char s1<height, width>;
        char s2<height, width>;
        char s3<height, width>;

        // Perform streamRead
        streamRead(s1, input0);
        streamRead(s2, input1);

        // Invoke kernel
        sum(s1, s2, s3);

        // Write results to output buffers
        streamWrite(s3, output0);

        // Check if there is any error
        if(s3.error())
        {
            printf("\nError occured %s\n", s3.errorLog());
        }
    }

    // Print the first output value
    printf ("\nThe Output: %c\n", output0[0]);

    // Free the allocated memory for inputs/output
    free (input0);
    free (input1);
    free (output0);

    return 0;
}
```

1.12 Complex Vector Constructor Usage

brcc allows vector constructors in any expressions.

The following examples show common usage cases.

```
    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 restriction that an N-component vector must have N components provided in its vector constructor. Each component can be a scalar constant or a 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
```

The ability to instantiate and initialize vectors using a vector constructors currently is supported only in kernel code.

Example:

```
kernel void multiply (float4 a, float4 b, out float4 c<>)
{
    c = a * b;
}

kernel void copy (float4 i0<>, out float4 o0<>)
{
   float2 index = indexof (o0).xy;
   float x = 1.f;
   float4 a = float4 (1.f, 1.f, 1.f, 1.f);
   float4 b = float4 (0.0f, 0.0f, 0.0f, 0.0f);
   float4 c = float4 (0.0f, 0.0f, 0.0f, 0.0f);

   multiply (float4 (x, 1.f, 1.f, x), float4 (1.f, 1.f, 1.f, 1.f), c);
   b = a - c + float4 (x, 1.f, 1.f, x) - float4 (1.f, 1.f, 1.f, 1.f) /
       float4 (x, 1.f, 1.f, x);
   o0 = i0 + b;
}

int main()
{
    unsigned int size = 1024;
    unsigned int i = 0, j = 0, mismatched = 0;
    unsigned int components = 4;
    float4 streami0<size>;
    float4 streamo0<size>;
    float *i0 = NULL;
    float *o0 = NULL;
    float *expectedo0 = NULL;

    i0 = (float*) malloc (size * sizeof (float) * components);
    o0 = (float*) malloc (size * sizeof (float) * components);
    expectedo0 = (float*) malloc (size * sizeof (float) * components);

    for (i = 0; i < size; ++i)
    {
        for (j = 0; j < components; ++j)
        {
            i0[i * components + j] = (float)i;
            expectedo0[i * components + j] = (float)i;
        }
    }

    streamRead (streami0, i0);
    copy (streami0, streamo0);
    if (streamo0.error ())
    {
        printf ("Error : %s", streamo0.errorLog());
    }
    streamWrite (streamo0, o0);
```

```
    for (i = 0; i < size; ++i)
    {
        for (j = 0; j < components; ++j)
        {
            if (o0[i * components + j] != expectedo0[i * components + j])
            {
                mismatched = 1;
                break;
            }
        }
        if (mismatched)
        {
            printf ("Failed", i);
            i = size;
            break;
        }
    }

    if (mismatched == 0)
    {
        printf ("Passed!!");
    }

    free (i0);
    free (o0);
    free (expectedo0);
}
```

1.13 Explicit Control Over Asynchronous APIs

Implicitly, Brook+ stream read requests and kernel invocations are asynchronous; stream write requests are synchronous. This implicit behavior lets Brook+ expose a very simple interface that allows the majority of developers to quickly implement algorithms in Brook+ without having to think about the actual hardware operations.

For certain applications, however, it is useful to provide explicit control over the asynchronous behavior of these requests. Thus, Brook+ lets developers force certain requests to be asynchronous, such as stream write calls. The developer then can overlap more operations on the CPU with operations on the GPU, achieving better overall system performance.

1.13.1 Usage

To request that a stream operation be issued asynchronously, add the `async` flag when issuing the command. By default, stream reads are asynchronous; thus, applying the `async` flag to stream reads has no effect.

Stream write requests issued with the `async` flag return immediately. The application must ensure all asynchronous operations on a stream are completed before trying to access the data pointer being written to by the stream. This can be done by waiting for the stream's `isSync()` method to return true, or by calling the stream's `finish()` method.

The status of kernel calls can be checked by checking the `isSync()` status on any of the output streams, or by calling `finish()` on any of the output streams.

Kernel calls and stream write operations implicitly synchronize on any required input stream data or kernel output data.

The Brook+ runtime can, at times, issue an asynchronous request synchronously if it determines that this more efficient.

1.13.2 Code

1.13.2 Examples

The following example shows how to use the asynchronous API to do CPU work in parallel

.

```
Stream<float> a (2, dim);
a.read (ptr);

while (!a.isSync())
{
    // CPU work
}

kernelCall (a);

while (!a.isSync())
{
    // CPU work
}

a.write (ptr, “async”);

while (!a.isSync())
{
    // CPU work
}
```

The following example shows how to write a tiled algorithm to leverage an asynchronous DMA and kernel call.

```
unsigned int numParts = 4;
Stream<float>** inStream = new Stream<float>*[numParts];
Stream<float>** outStream = new Stream<float>*[numParts];

// Declare these streams

inStream[0]->read (intile0);

////////////////////////////
// Next Kernel call and Data transfer from CPU->GPU is going to run in parallel
kernelCall (*inStream[0], *outStream[0]);
inStream[1]->read (intile1);
////////////////////////////

////////////////////////////
// Next three calls are going to run in parallel
outStream[0]->write (outTile0, "async");
kernelCall (*inStream[1], *outStream[1]);
inStream[2]->read (intile2);
////////////////////////////

////////////////////////////
outStream[0]->finish();
// Next three calls are going to run in parallel
outStream[1]->write (outTile1, "async");
kernelCall (*inStream[2], *outStream[2]);
inStream[3]->read (intile3);
////////////////////////////

////////////////////////////
outStream[1]->finish();
outStream[2]->write (outTile2, "async");
kernelCall (*inStream[3], *outStream[3]);
////////////////////////////

outStream[2]->finish();
outStream[3]->write (outTile3);
```

1.14 Memory Pinning

Memory pinning provides better performance for stream reads and stream writes. Brook+ implicitly handles most restrictions of the CAL memory pinning API.

1.14.1 Usage

To have a stream operation take advantage of memory pinning, add the flag `nocopy` when issuing the command.

1.14.2 Restrictions

- The base address of the application pointer must be 256-byte aligned.
- If `nocopy` is specified, and 'ptr' is not 256-byte aligned, the memory pinned read/write fails, and the control flow uses the existing non-memory pinned path without notifying the user.
- The buffer size should be a multiple of 64 bytes for best performance.
- If the width is not a multiple of 64 bytes, the memory pinned resource creation fails, and the non-memory-pinned method is used automatically.
- The maximum amount of memory that can be pinned is 4 MB. This amount may be reduced further, depending on the operating system and other system configuration factors.

1.14.3 Example Code

The following code example uses memory pinning for faster stream read/write.

```
#ifdef _WIN32
    ptr = _aligned_malloc (dim[0] * dim[1] * sizeof (float), 256);
#else
    cpuPtr = memalign (dim[0] * dim[1] * sizeof (float), 256);
#endif

Stream<float> a(2, dim);

// For memory pinned stream read
a.read (ptr, "nocopy");

while (!a.isSync())
{
    // CPU work
}

kernelCall(a);

while (!a.isSync())
{
    // CPU work
}

// Asynchronous memory pinned stream write
a.write (ptr, "async nocopy");

while (!a.isSync())
{
    // CPU work
}

#ifdef _WIN
32
    _aligned_free (ptr);
#else
    free (ptr);
#endif
```

1.15 brcc Preprocessor

Supported preprocessing directives are:

`#define`, `#undef`, `#ifdef`, `#ifndef`, `#else`, `#if`, and `#endif`

Unsupported preprocessing directives are:

`#elif`, `#include`, `#line`, `#error`, and `#pragma`

1.15.1 Syntax

Supported syntax for preprocessing directives are:

define

Syntax:

`#define` identifier replacement-list new-line

Replacement-list:

Single line statement

.

```
//! Example

kernel int4 multiply (int4 a, int4 b)
{
    return a * b;
}

#define MULTIPLY multiply (a, b)
#define SEVEN 7
#define INT4  int4 (1, 1, 1, 1)
kernel
void sum (int4 i0<>, int4 i1<>, out int4 o0<>)
{
    int4 a = int4 (SEVEN, 1, 1, 1);
    int4 b = INT4;

    o0 = i0 + i1 * MULTIPLY;
}

//! Line separator “\” not handled
//! Following is an error
#define MULTIPLY multiply (a, \
 b)
```

Example:

```
#define VALUE 10
#define VALUE 15

int main()
{
    int a = VALUE;
    return 0;
}
```

After running brcc:

```
int  main()
{
```

```
  int a = 15;
  return 0;
}
```

undef

Syntax: **#undef** identifier new-line

if

Syntax: **#if** integer-constant new-line group$_{opt}$
#if identifier new-line group$_{opt}$

An error results when:

- The integer-constant is not a 32 bit signed integer value
- The macro name is an identifier, and the macro value is not an integer-constant.

ifdef

Syntax:

#ifdef identifier new-line group$_{opt}$

ifndef

Syntax:

#ifndef identifier new-line group$_{opt}$

else

Syntax:

#else new-line group$_{opt}$

endif

Syntax:

#endif new-line

Note: Only comments ending on the same line as the preprocessor directive are supported on the same line as the preprocessor directive.

1.15.2 brcc Command Line Preprocessor Flags

1.15.2.1 –D Flag

Syntax:

a. `-D MACRO_NAME  //! To define macro`

Example:

```
-D WIN32
```

b. `-D MACRO_NAME=integer_constant //! To define macro with value`

Example:

```
-D VALUE=100 //! No spaces allowed between macro name and macro
          // value
```

c. `-D MACRO_NAME_ONE -D MACRO_NAME_TWO=10 //! Each macro must`
`// start with -D flag`

Example:

```
-D WIN32 -D VALUE=100
```

1.15.2.2 –pp Flag

This enables the preprocessor. By default, the preprocessor is not enabled.

1.16 Multi-GPU Support

The multi-GPU support in Brook+ allows the developer to query for what devices are available on the system and select which devices streams are allocated on and kernels are invoked on.

The multi-GPU implementation is thread safe and allows for different devices to be used concurrently in different threads of a single Brook+ program.

1.16.1 Usage

The following describes the APIs for accessing the multi-GPU feature in Brook+.

```
namespace brook
{
    // All devices available on the system - "all", "cpu", "cal"
    // Return list of devices and count
    Device* getDevices (const char* deviceType, unsigned int* count);

     // Use these devices for computation - Currently, only a single
     // device can be set per useDevices() call.
    Device* useDevices (Device* devices, unsigned int count,
                        unsigned int* oldDeviceCount);

    // Abstraction for different devices in system
    class Device
    {
        // Device type - Return "cal", "cpu"
        const char* getType() const;
    }
}
```

```
Device* brook::getDevices (const char* deviceType, unsigned int* count);
```

Input Parameter

`deviceType`: Developer can query different devices available on the system. Parameters can be “all”, “cal” and “cpu.” All other inputs return 0 devices.

Output Parameters

`count`: Total devices found that satisfy the given deviceType.

`return value`: Pointer to all the devices found on the system that satisfy the given deviceType.

The library destroys the returned pointer; the application must not try to delete it.

```
Device* brook::useDevices (Device* devices, unsigned int count,
                           unsigned int* oldDeviceCount);
```

Input Parameter

`devices`: All the devices to be used for computing.

`count`: Number of devices to use in the given devices list.

Output Parameter

Information about older devices. The developer can retrieve this information when changing device state; this also can be used to revert the state.

`oldDeviceCount`: Returns the number of devices set earlier. This parameter is optional and NULL can be passed.

Return value: Devices set earlier. If `oldDeviceCount` is NULL, the return value also is NULL.

1.16.1.1 Notes on `useDevices()`

After calling `useDevices()`, all declared streams or kernel invocations use the given device for computation. It can be called multiple times in the same thread.

Stream allocations and kernel invocations must occur on the same device. This requirement is checked at runtime.

It is possible to write a multi-threaded program that uses all available devices concurrently. The device state must be set inside the thread; setting a device in one thread does not affect the device state in another thread. Child threads spawned from a parent thread do not inherit the device settings of the parent thread.

Currently, it is not possible for multiple threads to work concurrently with the same device. It is the application’s responsibility to serialize such calls.

1.16.1.2 Backward Compatibility

If `useDevices()` is not used in a program or a thread, a default device is used (the first CAL device). This can be changed by using the environment variable BRT_RUNTIME or BRT_ADAPTER.

1.16.2 Sharing Data Between Different Devices

If the application must share data between kernels on different devices, it can use the `assign()` method available in the Stream class. The `assign()` call is issued asynchronously; see Explicit Control Over Asynchronous APIs, for information on explicit controls over asynchronous APIs.

1.16.3 Example Code

The following program demonstrates multi-GPU API usage

.

```
// get all the devices available on the system
unsigned int deviceCount = 0;
Device* deviceList = brook::getDevices ("all", &deviceCount);

for (unsigned int i = 0; i < deviceCount; ++i)
{
    // I want to use cal device
    if (!strcmp(deviceList[i].getType(), "cal"))
    {
        // Use this device
        brook::useDevices (deviceList + i, 1, NULL);

        // declare stream
        brook::Stream<float> a(rank, dim);

        // Call same kernel on multiple devices
        kernelCall(a);

        // Write to the pointer
        a.write (ptr);
    }
}
```

The following program has multiple kernels executing in parallel

.

```
// get all the CAL devices available on the system
unsigned int deviceCount = 0;
Device* deviceList = brook::getDevices ("cal", &deviceCount);

// declare streams
brook::Stream<float>** a = new brook::Stream<float>*[deviceCount];

for (unsigned int i = 0; i < deviceCount; ++i)
{

    // Use this device
    brook::useDevices (deviceList + i, 1, NULL);

    a[i] = new brook::Stream<float>(rank, dim);
}
// Run kernels in parallel - leverage Asynchronous nature
// kernel call
for (unsigned int i = 0; i < deviceCount; ++i)
{
    // Get Device information from stream
    unsigned int count = 0;
    Device* devices = a[i].getDevices (&count);
    brook::useDevices (devices, count, NULL);

    kernelCall (a[i]);
}
```

The following code uses multiple devices concurrently in a multi-threaded program

.

```
struct ThreadData
{
    brook::Device* device;
    float* outData;
};

unsigned int __stdcall run (void* data)
{
    ThreadData* threadData = (ThreadData*)data;

    brook::useDevices (threadData->device, 1, NULL);

    brook::Stream<float> outputStream (rank, dim);
    kernelCall (outputStream);
    outputStream.write (threadData->outData);

    return 0;
}
unsigned int count = 0;
brook::Device* device = brook::getDevices ("all", &count);

ThreadData* data = new ThreadData[count];
HANDLE* hThread = new HANDLE[count];
unsigned int* threadID =  new unsigned int[count];

// Initialize thread data

for (unsigned int i = 0; i < count; ++i)
{
    hThread[i] = (HANDLE)_beginthreadex (NULL, 0, &run, (void*)(data + i),
                  0, threadID + i);
}

for (unsigned int i = 0; i < count; ++i)
{
    WaitForSingleObject (hThread[i], INFINITE);
}

delete[] data;
delete[] hThread;
delete[] threadID;
```

1.17 Thread Data Sharing

The current generation of AMD GPUs allows threads within a single group to share data and synchronize with each other. This can be useful in certain applications where inter-thread communication is either vital to the algorithm or can vastly speedup the execution of the application.

Brook+ exposes this capability through:

- attributes for specifying group sizes,
- syntax for specifying a shared buffer, and
- defined semantics for reading from, and writing to, the shared buffer.

A mechanism is also provided to synchronize execution between threads within a group.

1.17.1 Specifying Thread Count in a Group

You can specify the number of threads within a group by specifying an attribute for the kernel. To do this, use `Attribute[GroupSize (x, y, z)]`, where `x`, `y`, and `z` are the number of threads in a 3D group in the X, Y, and Z directions, respectively.

The following demonstrates both valid and invalid attribute descriptions.

```
Attribute[GroupSize (256, 1, 1, 1)]    //! Valid and is equal to Attribute[GroupSize (64)]
Attribute[GroupSize (64, 2, 1, 1)]     //! Invalid
                                     //! Higher dimension values must
                                        be 1, if specified
```

The maximum group size is 1024. Values greater than 1024 result in a compilation error.

Group size is obtained by multiplying the number of threads in the group in the X, Y, and Z direction.

It is illegal to specify a GroupSize for kernel with an ordinary output

.

1.17.2 Representing Shared Memory

To declare a shared memory array in a kernel, use the `shared` keyword before declaring an array. The following illustrates the syntax used to declare a shared memory array.

```
shared float4 lds[size];
```

The declared array size, *size*, is the shared memory array size for an entire group of threads. The array name can be any valid variable name in Brook+.

There are two restrictions when declaring a shared memory array:

- The data type size must be 128 bits.
- The size must be a constant integer, and cannot be a constant expression.

The following examples illustrate valid and invalid shared memory array declarations.

```
shared float4 lds[256];                                    //! Valid

shared float4 lds[256 * 2];                                   //! inValid

const int size = 256;
shared float4 lds[size];                                   //! inValid
```

It is illegal to declare a shared memory array in a kernel without an attached `GroupSize` attribute.

1.17.3 Relationship Between Group Size and Shared Memory Array Size

The allocated shared memory array is divided equally among the threads in a group. The application must guarantee that the total shared memory array size is an integer multiple of the group size.

Each thread in a group effectively owns its equal share of the total shared memory array. Each thread's share of the shared memory array must be a multiple of 128-bits and be no larger than 64 bytes.

1.17.4 Obtaining a Thread's Group Offset

It may be necessary for a thread to determine its offset within its thread group. To do this, a thread calls the `instanceInGroup()` function in the kernel. This function works similarly to `instance()`. It returns a four-integer vector indicating the threads location in the multi-dimensional thread group space:

```
int4 item = instanceInGroup();
```

1.17.5 Accessing Shared Memory

Brook+ allows the application to access the shared memory array in a manner similar to C-style array indexing.

1.17.5.1 Shared Memory Read

The following code is an example of a read.

```
data = lds[index];  //! LDS read
```

A thread is allowed to read from any location within the shared memory array. If the location is owned and written to by another thread, proper synchronization techniques must be used to prevent undefined results (see Synchronization).

1.17.5.2 Shared Memory Write

1.17.6 A thread can write to any location within the shared memory array that it owns. See Relationship Between Group Size and Shared Memory Array Size

, for more information about how to calculate what part of the shared memory array a particular thread owns.

To enforce the owner-write model, all shared memory writes must be done using the following syntax.

```
lds[Stride * threadId + offset] = data;  //! LDS write
```

where: `Stride` = an integer constant (SharedArraySize/GroupSize)
`threadId` = a value returned by `instanceInGroup().x`

`offset` = integer constant and
0 ≤ offset < (SharedArraySize/GroupSize)

If any of these three parameters is missing from the index expression, a compiler error results. Attempting to write to a portion of the shared memory array owned by another thread produces undefined results.

1.17.7 Synchronization

`syncGroup()` allows the application to synchronize all threads in a group. When all threads in a group have executed `syncGroup()` synchronized, execution can proceed past the `syncGroup()` call.

Note that `syncGroup()` can be used inside a conditional statement, but only if the conditional expression evaluates identically across the entire group; otherwise, code execution is likely to hang or produce undefined results.

`syncGroup()` can be used to synchronize access to shared memory and global memory.

1.17.8 Example Code

The following example illustrates thread data sharing.

```
Attribute[GroupSize (64, 1, 1)]
kernel void sum_2d (float4 a<>, float4 b<>, out float4 c[][])
{
   shared float4 lds[256]; // defines the amount of data per thread as 4
                              float4 (256/64)

   int2 index = instance ().xy;
   int item  = 0;

   lds[4 * instanceInGroup ().x + 2] = a + b; // write to data of thread
                                                 id item and at offset 2.
   syncGroup();
   item = 4 * instanceInGroup ().x + 2;
   c[index.y][index.x] = lds[item ];   // read data of thread id (index/4)
                                          and at offset index%4.
}

int main()
{
    unsigned int width = 1024;
    unsigned int height = 1024;
    unsigned int i = 0, j = 0, k = 0, mismatched = 0;
    unsigned int components = 4;
    float4 streami0<height, width>;
    float4 streami1<height, width>;
    float4 streamo0<height, width>;
    float *i0 = NULL;
    float *i1 = NULL;
    float *o0 = NULL;
    float *expectedo0 = NULL;

    i0 = (float*) malloc (height * width * sizeof (float) * components);
    i1 = (float*) malloc (height * width * sizeof (float) * components);
    o0 = (float*) malloc (height * width * sizeof (float) * components);
expectedo0 = (float*) malloc (height * width * sizeof (float) *
     components);

    for (i = 0; i < height; ++i)
    {
        for (j = 0; j < width; ++j)
        {
            unsigned int index = i * width + j;
            for (k = 0; k < components; ++k)
            {
                i0[index * components + k] = (float)i;
                i1[index * components + k] = (float)j;
                expectedo0[index * components + k] = (float)(i + j);
            }
        }
    }

    streamRead (streami0, i0);
    streamRead (streami1, i1);
    sum_2d (streami0, streami1, streamo0);
    if (streamo0.error())
        {
        printf ("Error : %s", streamo0.errorLog());
        }
    streamWrite (streamo0, o0);
    free (i0);
    free (i1);
    free (o0);
    free (expectedo0);
}
```

1.18 DX Interoperability

DX interoperability is an important feature for many applications, such as image/video processing. It enables the application to render data after kernel execution without needing to read the generated data back to the CPU. This interoperability has the following features.

- The developer can query if DX interoperability is supported on a particular device.
- The application can process graphics resource data in the same manner it processes normal stream data.

1.18.1 Usage

The following describes the APIs for accessing the DX interoperability feature in Brook+.

```
class Device
{
  // Supported device bindings
  // return a space separated list of all supported bindings
  // e.g. “d3d9 d3d10”
  const char* getBindings()const;
}
```

Before trying to interoperate with DX, the application must first call `getBindings()` to determine if the DX version used is supported by Brook+. DX interoperability is currently only supported on Microsoft Windows Vista. DX interoperability is not yet supported on Microsoft Windows XP or Linux.

To share data between Brook+ and DX, the application must instantiate a Stream object and pass in the appropriate binding type as an argument to the constructor.

Once a Stream object has been created with the appropriate binding type, the application must use the stream's `setProperty()` method to set all of the necessary properties before issuing any operations on the stream. Failure to properly set all properties before a stream is used results in an error. Passing in invalid values for the properties, such as NULL, results in an error.

Table 1.3 DX Interoperability Properties describes the three properties that must be set for DX interoperability and the equivalent values in the DX API.

Table 1.3 DX Interoperability Properties

Property Name	D3D9 Value	D3D10 Value
d3dDevice	IDirect3DDevice9* d3dDevice	ID3D10Device* d3dDevice
d3dResource	IDirect3DResource9* resource	ID3D10Resource* resource
shareHandle	Handle handle	Handle handle

The DX resource being shared with Brook+ must reside on the same device as the kernel invocation that uses the data.

The application must create all pointers and buffers passed to the stream with `setProperty()`. After the stream in no longer in use, the application must free all pointers and buffers that were passed to the stream.

1.18.2 Example Code

The following example code shows how to use DX interoperability in Brook+ with D3D9.

```
void func (IDirect3DDevice9* d3dDevice, IDirect3DResource9* d3dResource,
          HANDLE handle)
{
    unsigned int deviceCount = 0;
    Device* deviceList = brook::getDevices ("cal", &deviceCount);

    // Get all the extensions supported by first CAL device
    std::string extensions (deviceList[0].getBindings());

    // Check if d3d9 extension is supported
    if (extensions.find ("d3d9") != std::string::npos)
    {
        // Use first CAL device
        brook::useDevices (deviceList, 1);

        Stream<float> outStream (rank, dim, "d3d9");
        outStream.setProperty ("d3dDevice", (void*)d3dDevice);
        outStream.setProperty ("d3dResource", (void*)d3dResource);
        outStream.setProperty ("shareHandle", (void*)handle);


        // Was stream creation successful
        if (outStream.error())
        {
            std::cout << outStream.errorLog();
        }

        // Call a kernel
        imageProcessKernel (outStream);

        // Render output resource
        display (d3dResource);
    }
}
```