AMD

Smarter Choice

# High Level Programming for GPGPU

Jason Yang
Justin Hensley

# Outline

Brook+

Brook+ demonstration on R670

AMD IL

# Brook+ Introduction

# What is Brook+?

Brook is an extension to the C-language for stream programming originally developed by Stanford University.

Brook+ is an implementation by AMD of the Brook GPU spec on AMD's compute abstraction layer with some enhancements.

# Simple example

```
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);
    ...
}
```

# Simple example

```
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);
    ...
}
```

**Kernels** - Program functions that operate on streams

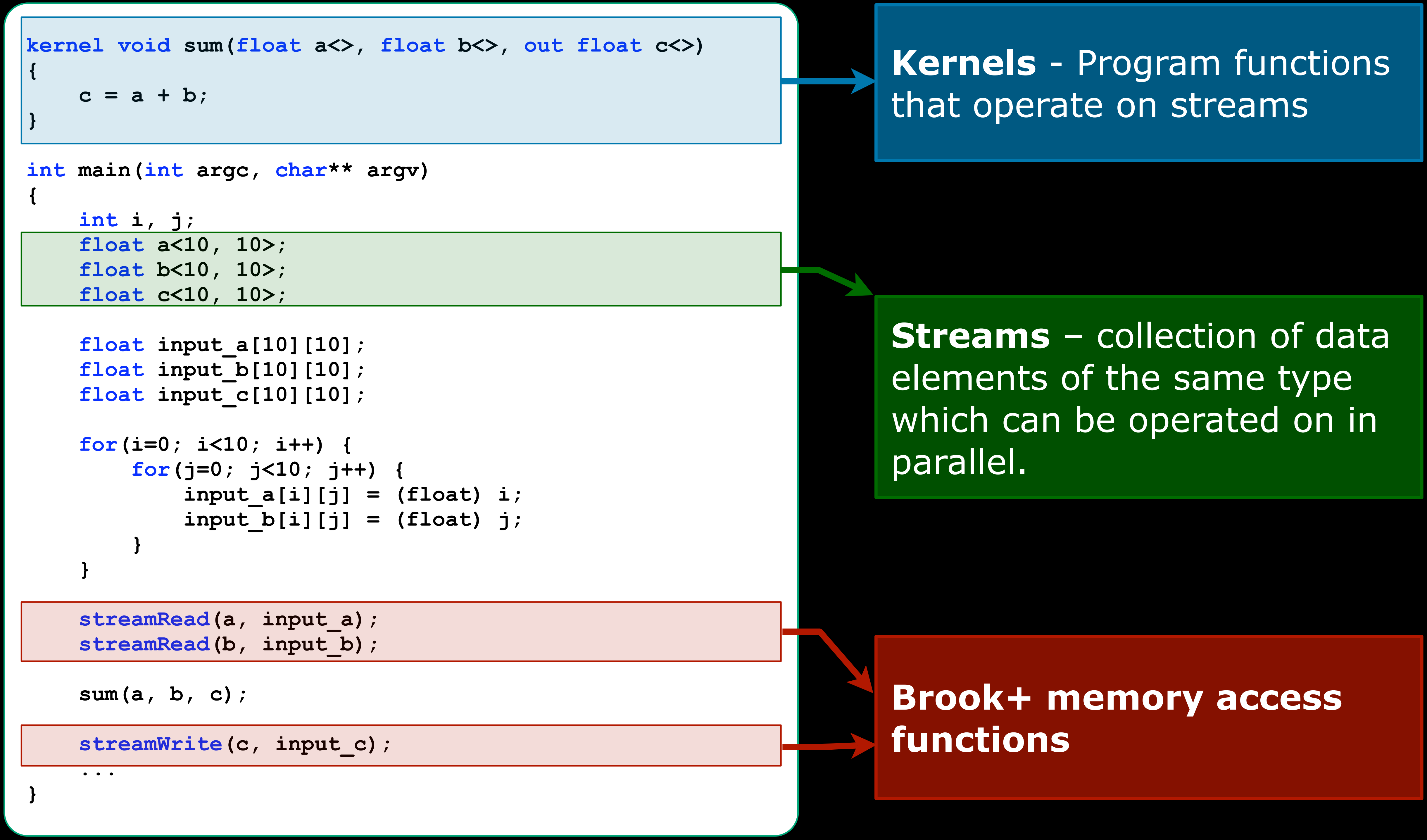
# Simple example

```
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);
    ...
}
```

**Kernels** - Program functions that operate on streams

**Streams** – collection of data elements of the same type which can be operated on in parallel.

# Simple example



```
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);
    ...
}
```

**Kernels** - Program functions that operate on streams

**Streams** – collection of data elements of the same type which can be operated on in parallel.

**Brook+ memory access functions**

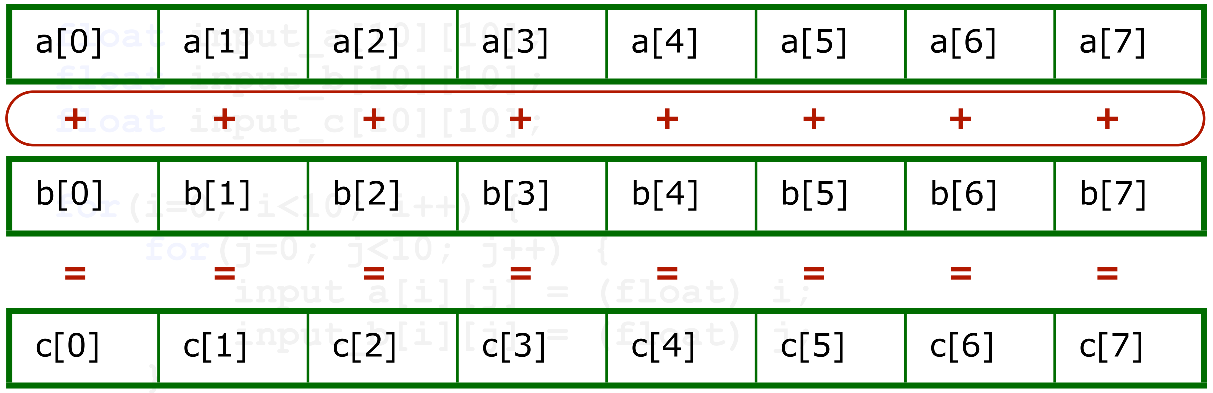
# What’s the idea of stream computing?

Execute programs (kernels) on each element of an input data array (streams) and outputting the result to another array.

- –Data parallelism
- –Transparent access to the processing cores

# Stream computing

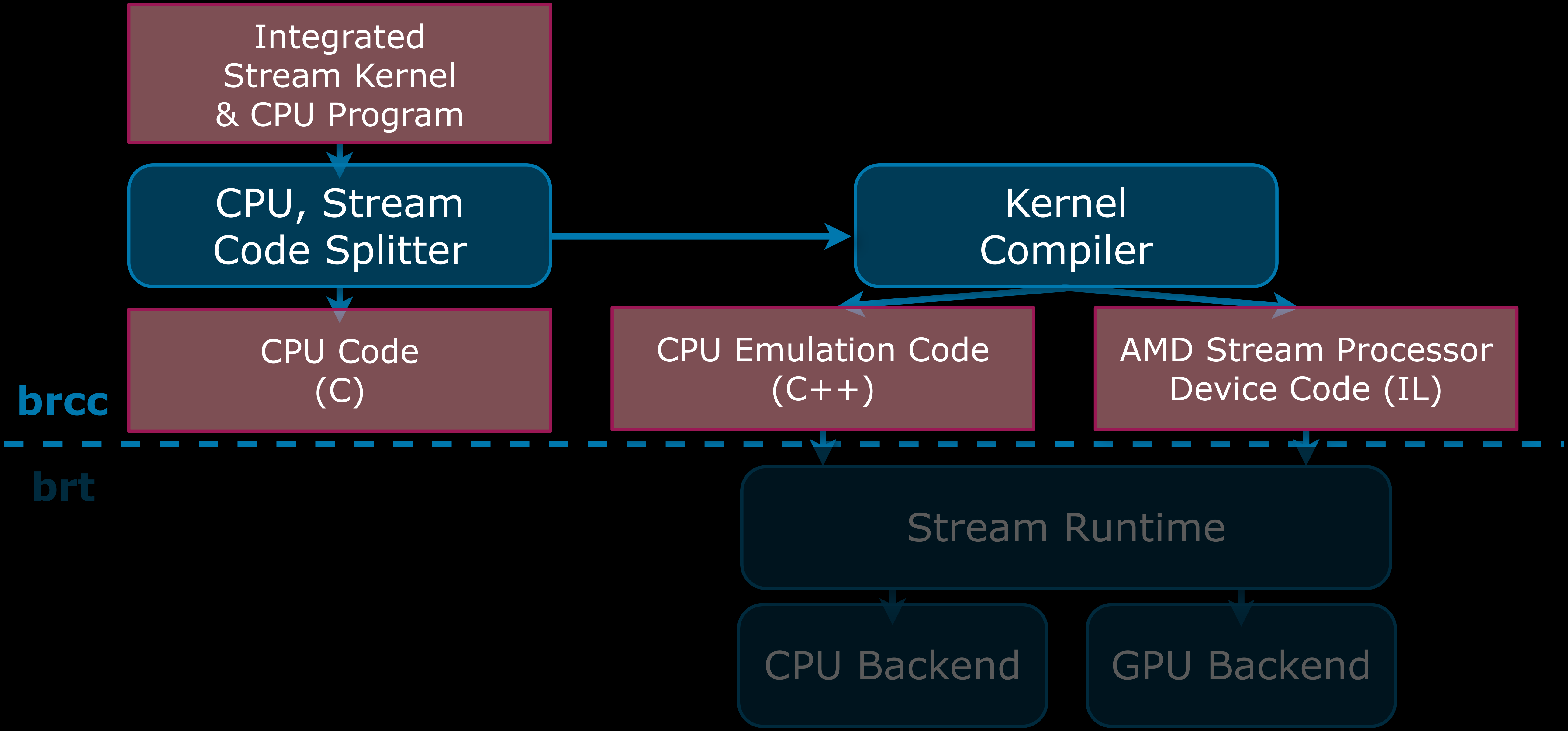
```
kernel void sum(float a<>, float b<>, out float c<>)
{
    c = a + b;
}

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);
    ...
}
```



| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |
|---|---|---|---|---|---|---|---|
| + | + | + | + | + | + | + | + |
| b[0] | b[1] | b[2] | b[3] | b[4] | b[5] | b[6] | b[7] |
| = | = | = | = | = | = | = | = |
| c[0] | c[1] | c[2] | c[3] | c[4] | c[5] | c[6] | c[7] |

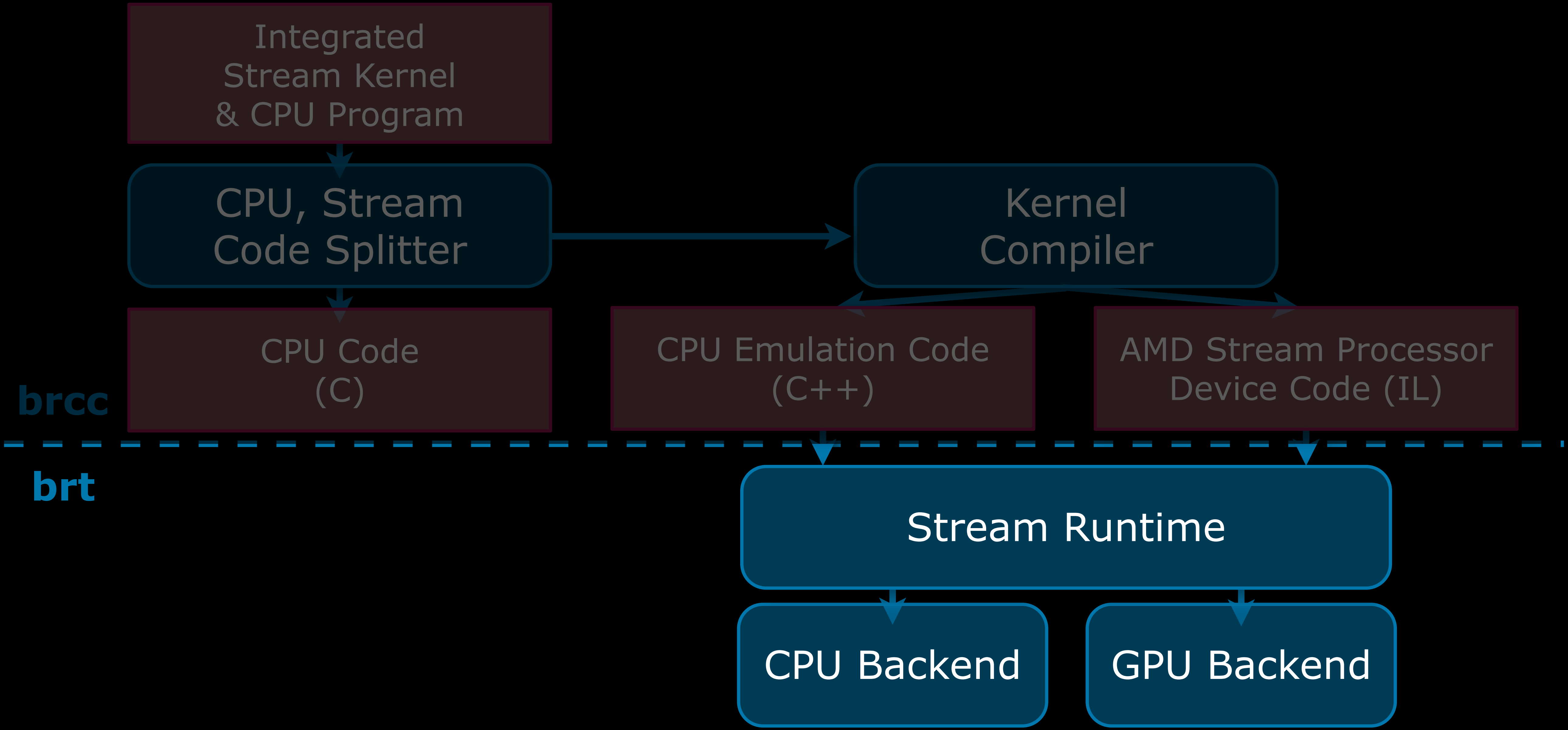
# Brook+ Compiler

Converts Brook+ files into C++ code. Kernels, written in C, are compiled to AMD’s IL code for the GPU or C code for the CPU.


Integrated Stream Kernel & CPU Program
CPU, Stream Code Splitter
Kernel Compiler
CPU Code (C)
CPU Emulation Code (C++)
AMD Stream Processor Device Code (IL)
brcc
brt
Stream Runtime
CPU Backend
GPU Backend

# Brook+ Runtime

IL code is executed on the GPU.  The backend is written in CAL.


Integrated Stream Kernel & CPU Program
CPU, Stream Code Splitter
Kernel Compiler
CPU Code (C)
CPU Emulation Code (C++)
AMD Stream Processor Device Code (IL)
brcc
brt
Stream Runtime
CPU Backend
GPU Backend

# Brook+ features today (1.0 Alpha)

Brook+ is an extension to the Brook for GPUs source code (open source).

Features of Brook for GPUs relevant to modern graphics hardware are maintained.

Kernels are compiled to AMD’s IL.

Runtime uses CAL to execute on AMD GPUs.

- CAL runtime generates ASIC specific ISA dynamically

Original CPU backend also included.

- Currently used mainly for debugging
- Optimizations currently underway

# Brook+ coming *very* soon (1.0 Beta)

Double precision

Scatter (mem-export)

Graphics API interoperability

- currently *readback* required

Multi-GPU support

Linux, Vista, XP

- 32 & 64-bit

Extension Mechanism

- Allow ASIC specific features to be exposed without ‘sullying’ core language

# Brook+ Language

# Writing Brook+ code

What do you use the Brook+ language for?

- Brook+ kernels
- Executing Brook+ kernels
- Stream handling code
  - Reading and writing user data into streams

Application code can be written in Brook+, but is not necessary

# Brook+ is based on C

```
int main(int argc, char** argv)
{
    float input_a[10];
    float input_b[10];

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

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

    ... //Do GPU Stuff

}
```

What’s wrong with this code?

# Brook+ is based on C

What’s wrong with this code?

```
int main(int argc, char** argv)
{
    float input_a[10];
    float input_b[10];

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

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

    ... //Do GPU Stuff

}
```

**Variables cannot be declared inline; only at beginning of code blocks**

# Preprocessor caveats

Brook+ compiler has no built-in preprocessor

If the kernel has preprocessor directives, it must be processed before handing it to the compiler

Preprocessor directives in non-kernel code are passed through to the subsequent compiler stages

# Short vector types

Standard C types supported with exceptions for streams and kernels

Short vectors (2 to 4 elements) used similarly to shader programming

- names built by appending the number to the type (e.g., “int2”, “float4”)
- doubles are limited to up to 2 elements
- access to individual fields is through structure member syntax: “.x”, “.y”, “.z”, “.w”
- fields can be accessed in any order and combination up to four fields (e.g., “.xyzw”, “.xxx”, “.zwy”)
- applying an operator to operands of vector types is equivalent to applying the operator to each field individually

# Basic Brook+ code

Streams

Kernels

Kernel execution

This talk will focus on 1.0 Beta (soon to be released)

# Streams

# Streams

Data arrays that are operated on by kernels

In the GPU context they are data arrays or texture surfaces that reside in GPU local memory

Streams elements all have the same type

- –can use a struct for multiple types in a stream

# Streams (cont.)

Limitations

- GPU hardware only natively support sizes of 8192x8192
- Brook+ can support larger sizes using software address translation, which could degrade performance
- With address translation largest 1D array is $2^{26}$

Supported types

- float and float vector
- double and double vector
- structs of float and double types

# Declaring streams

Similar to C style array declaration except angle brackets are used in place of square brackets

- *type name<n>*; //1D stream array of *type* with size *n*
- *type name<n, m>*; //2D stream array of *type* with dimensions *nxm*

Examples cases:

float a<5>;

float b<2, 3>;

double c<3>[5]; // array of streams

double d[3]<5>; // stream of arrays

# Dynamic allocation

There is no equivalent malloc() function for streams

Problem: How to dynamically create streams in Brook+ if declarations cannot be inlined?

# Dynamic allocation

There is no equivalent malloc() function for streams

Problem: How to dynamically create streams in Brook+ if declarations cannot be inlined?

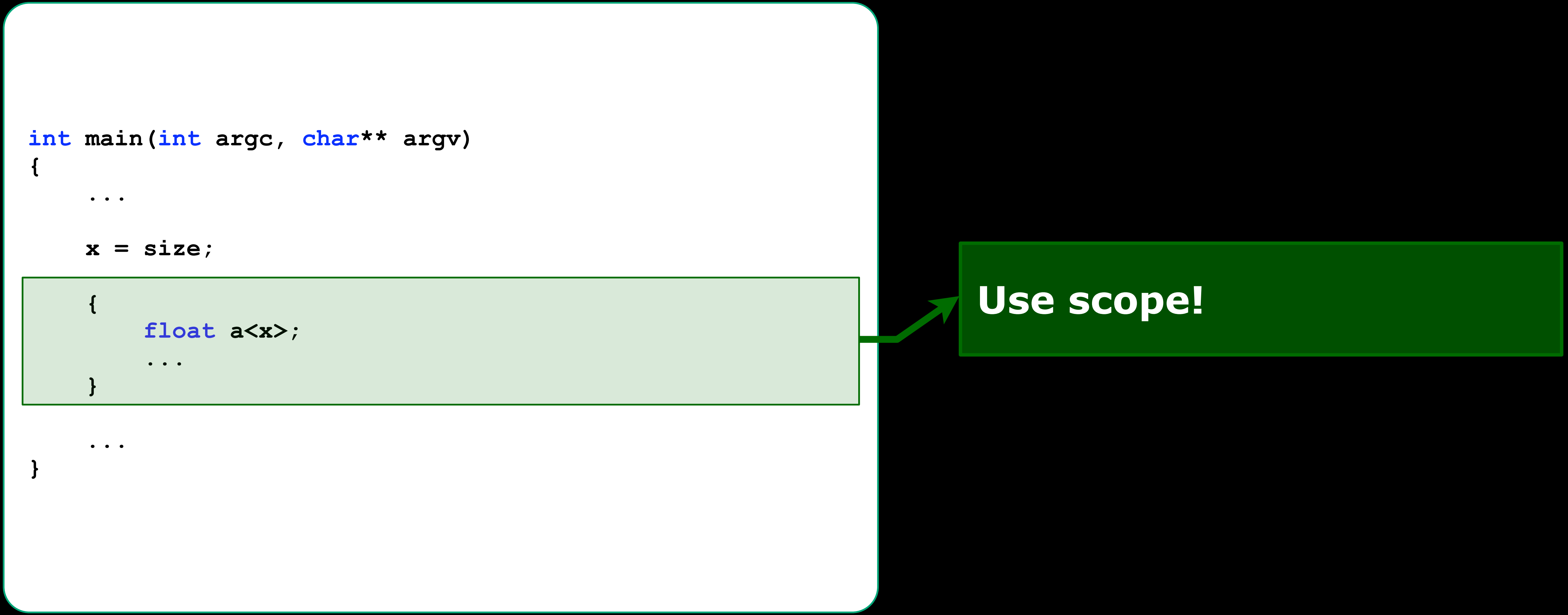
```
int main(int argc, char** argv)
{
    ...

    x = size;

    float a<x>;

    ...

}
```

**Invalid! Variables must be declared at beginning of code blocks**

# Dynamic allocation

There is no equivalent malloc() function for streams

Problem: How to dynamically create streams in Brook+ if declarations cannot be inlined?



```
int main(int argc, char** argv)
{
    ...

    x = size;

    {
        float a<x>;
        ...
    }

    ...
}
```

**Use scope!**

# Accessing streams

Streams cannot be directly accessed by the user (i.e. you cannot read/write stream elements)

All interaction must be done through IO stream operators

streamRead(*destination_stream*, *source_array*)

- copies data from the *source_array* to the *destination_stream*
- in the GPU context, data is copied from CPU memory to GPU memory

streamWrite(*source_stream*, *destination_array*)

- copies data from the *source_stream* to the *destination_array*
- in the GPU context, data is copied from GPU memory to CPU memory

User is responsible for input dimensions to match

# How streams are handled by CAL

On stream creation/deletion a buffer is allocated/ deallocated on the GPU with little overhead

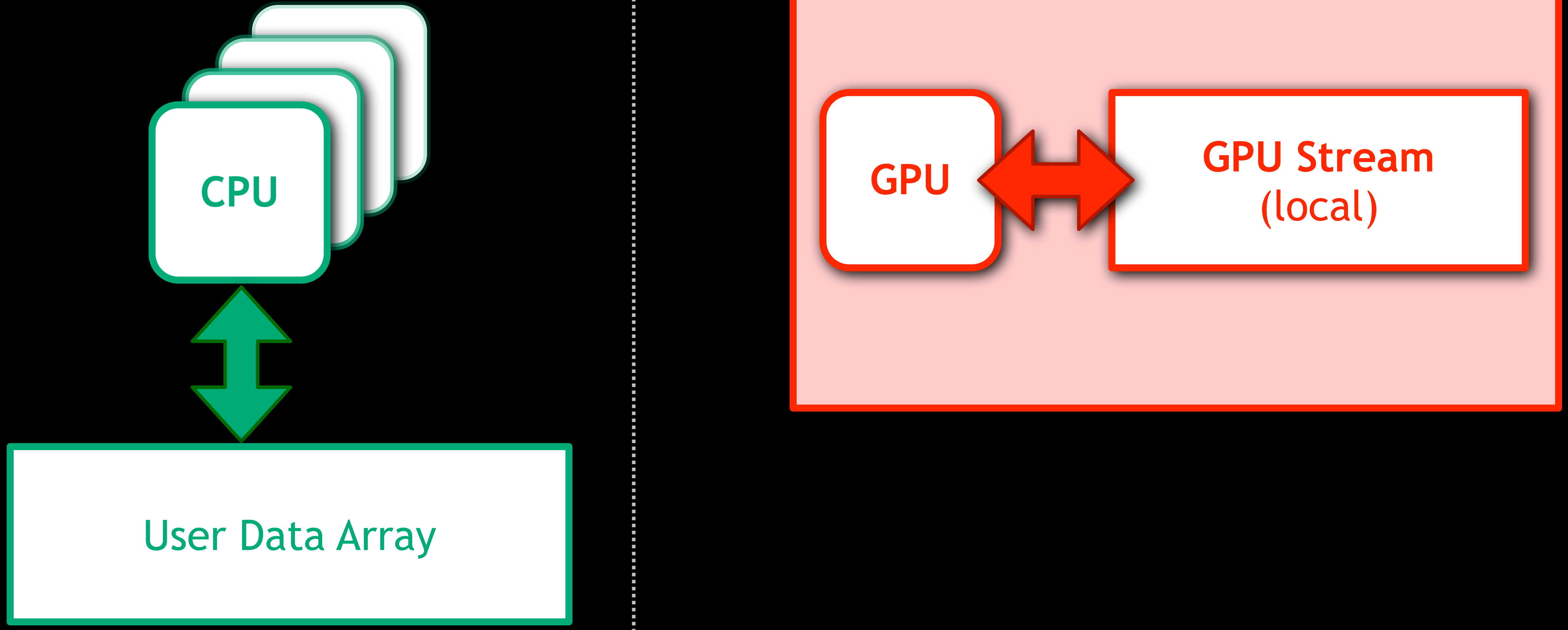
On streamRead or streamWrite, data is not immediately copied

streamRead operation

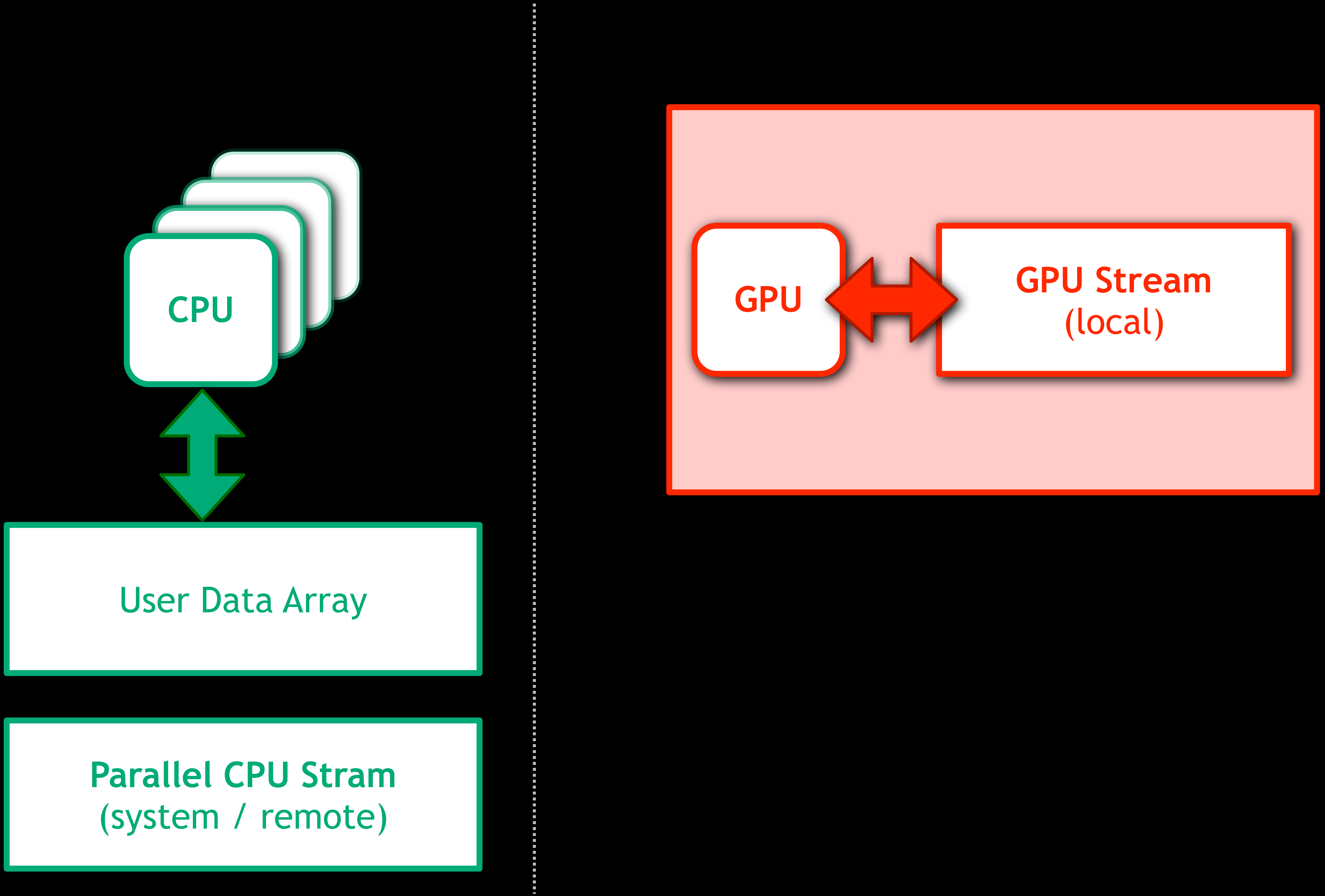
- Parallel CPU stream array is created and locked from further access
- data is copied from the user array to a parallel CPU stream array and then unlocked
- data is then asynchronously transferred from CPU to GPU and completion is signaled

The reverse happens for streamWrite

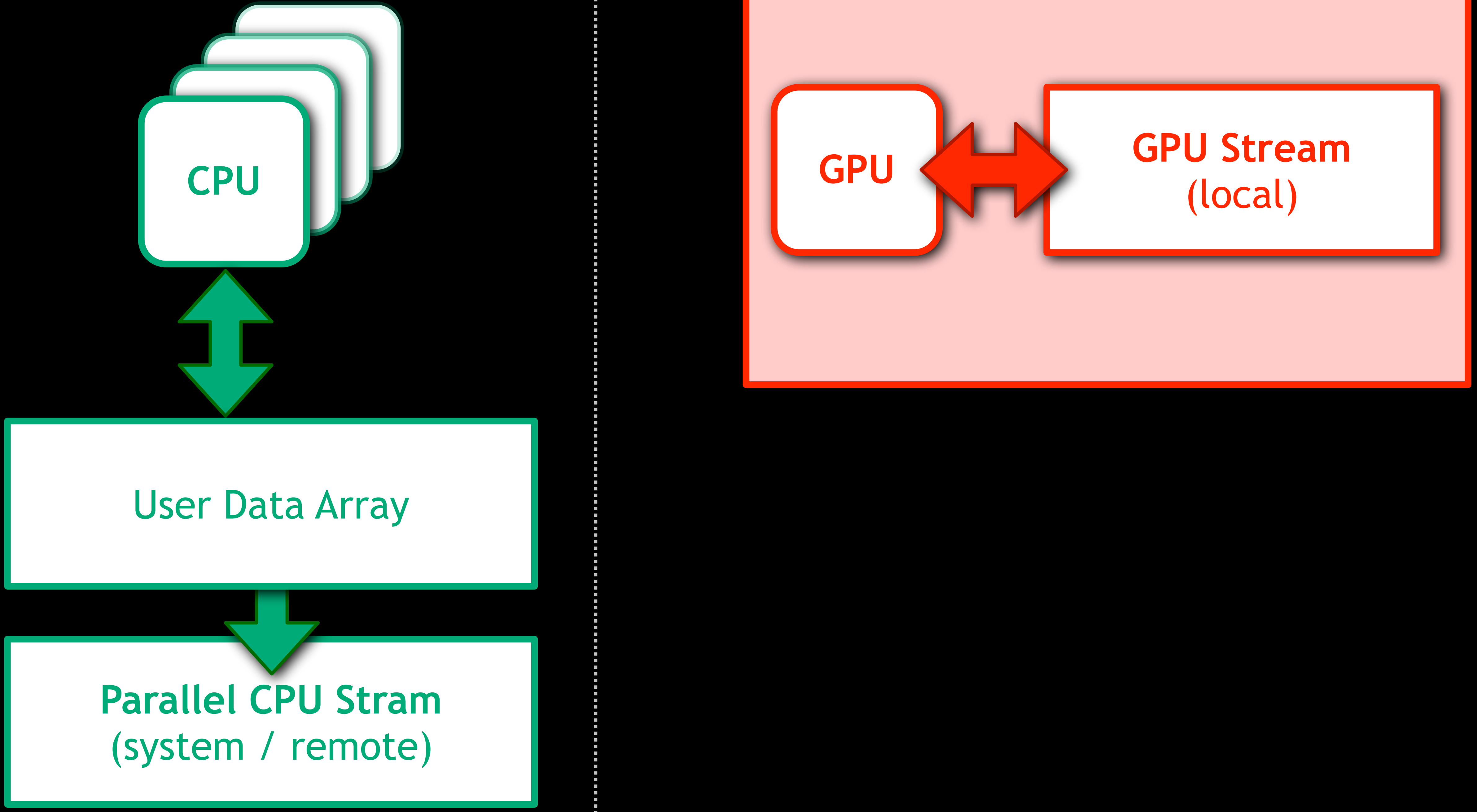
# streamRead()



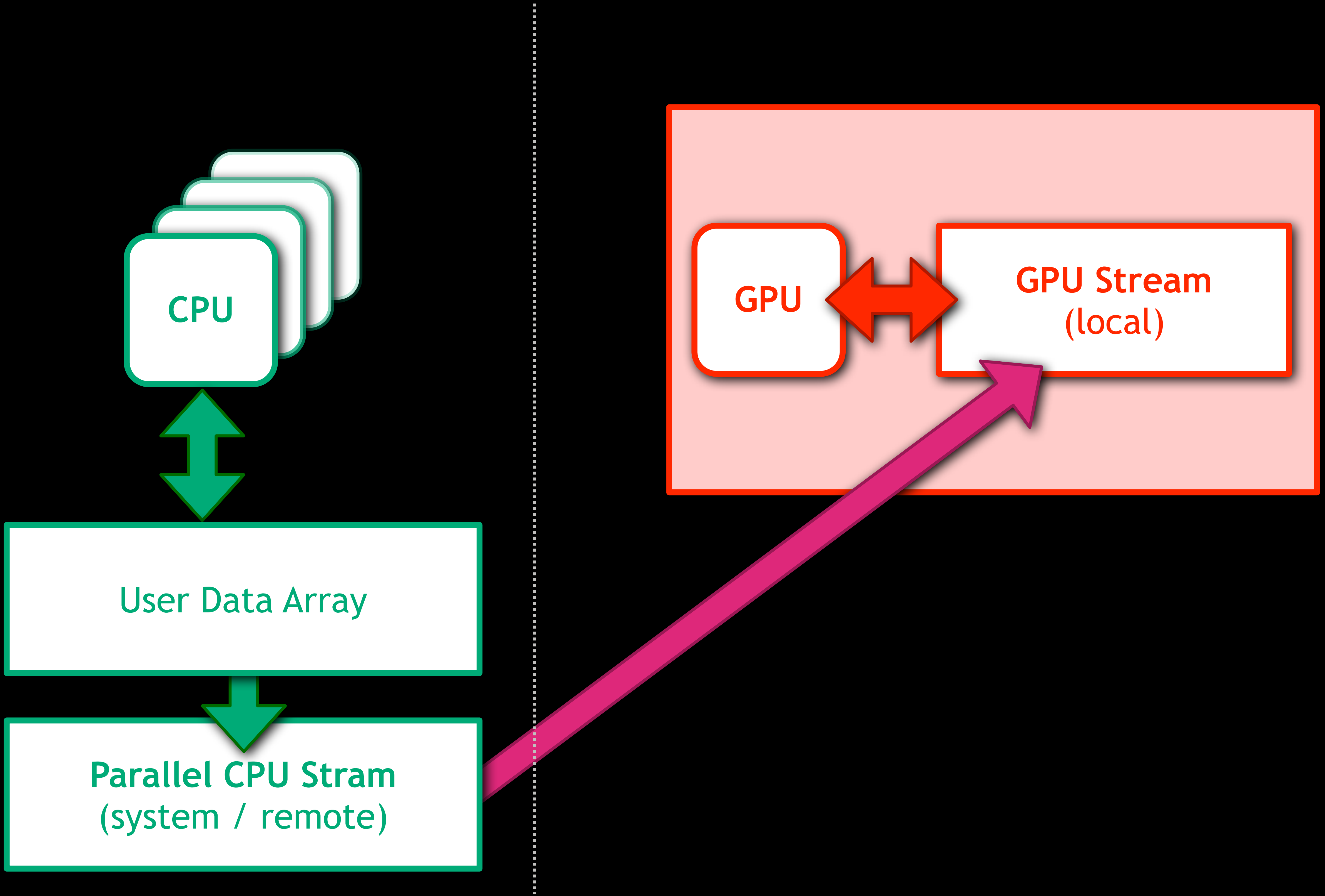
# streamRead()


CPU
User Data Array
Parallel CPU Stram
(system / remote)
GPU
GPU Stream
(local)

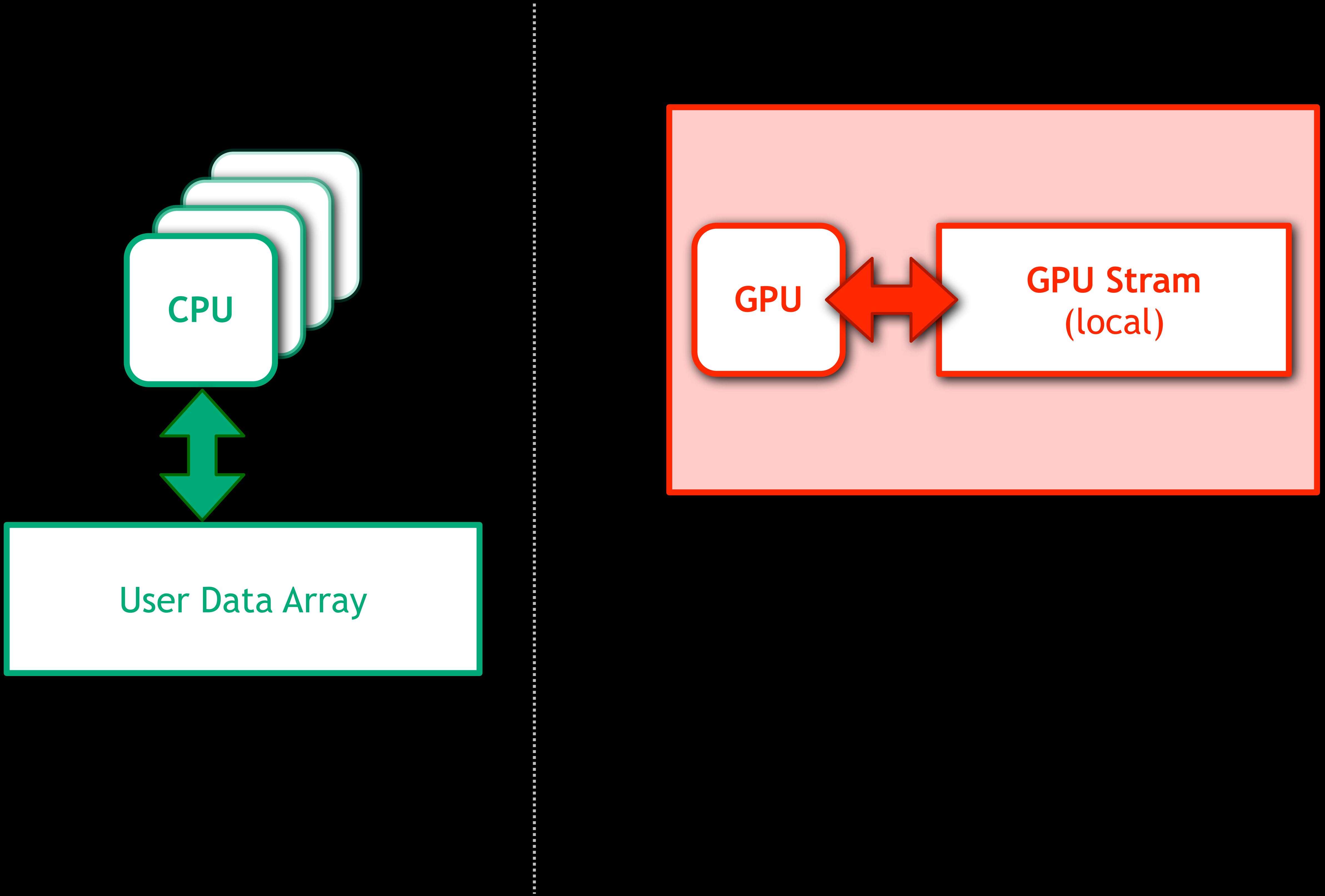
# streamRead()


CPU
User Data Array
Parallel CPU Stram
(system / remote)
GPU
GPU Stream
(local)

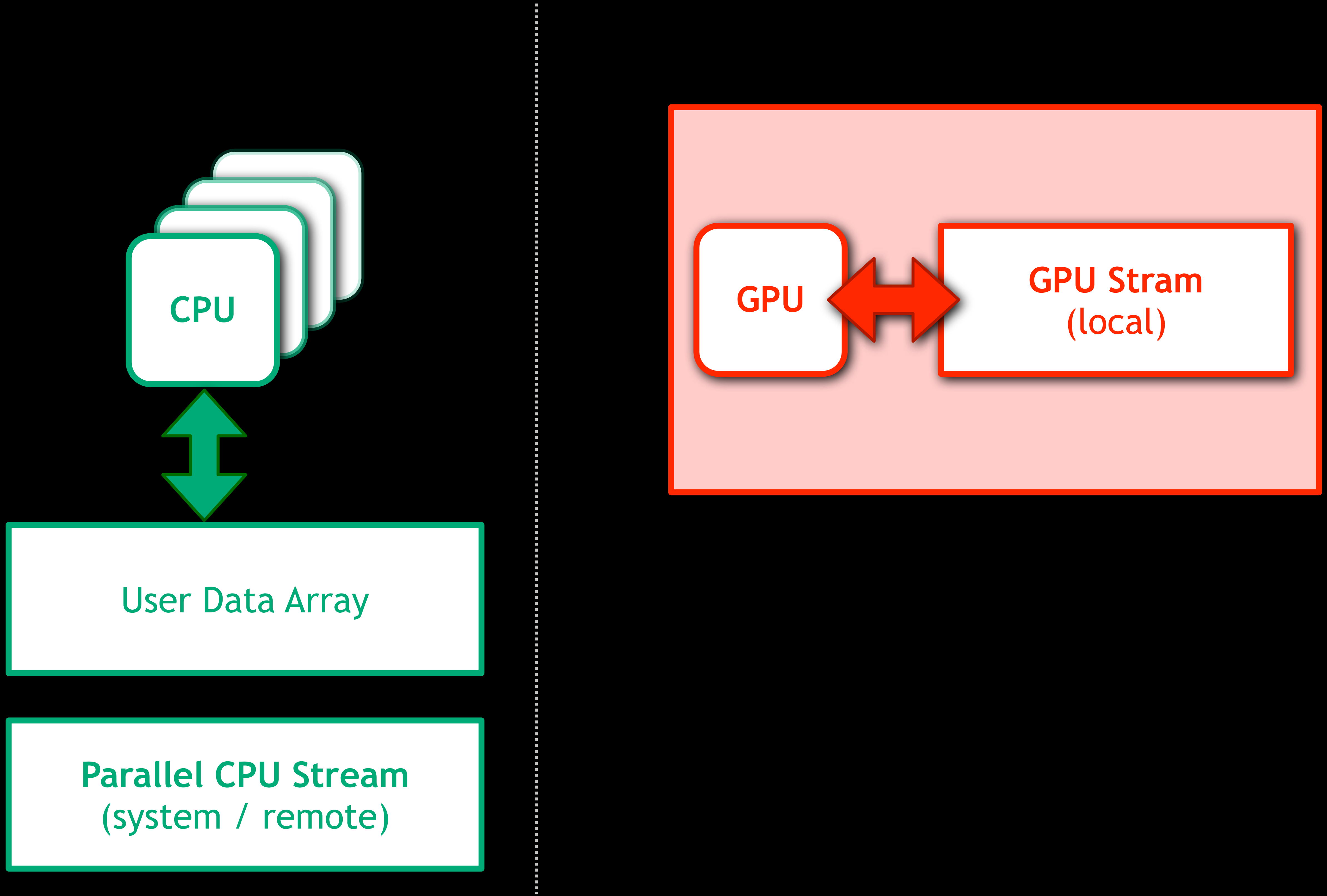
# streamRead()


CPU
User Data Array
Parallel CPU Stram
(system / remote)
GPU
GPU Stream
(local)

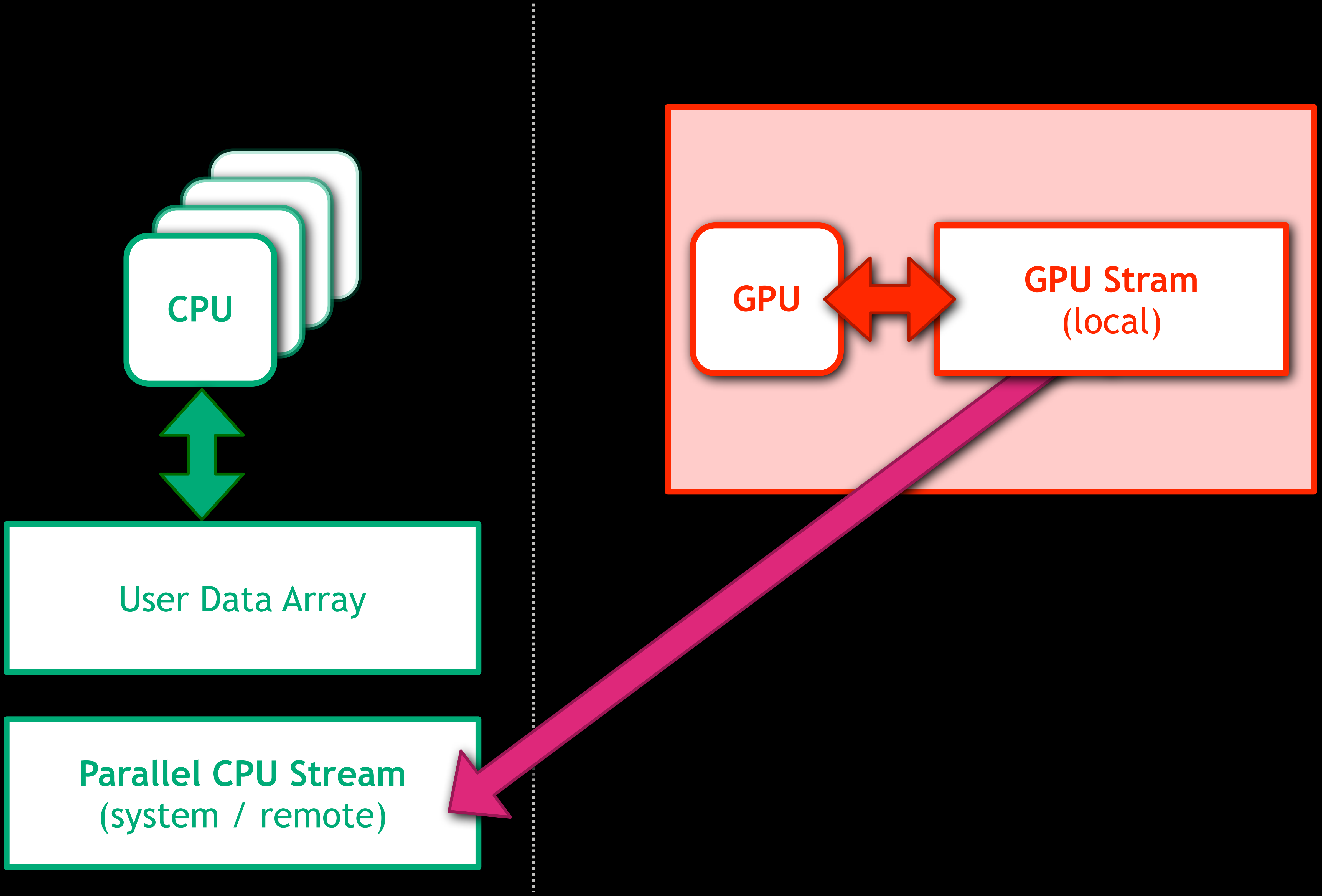
# streamWrite()



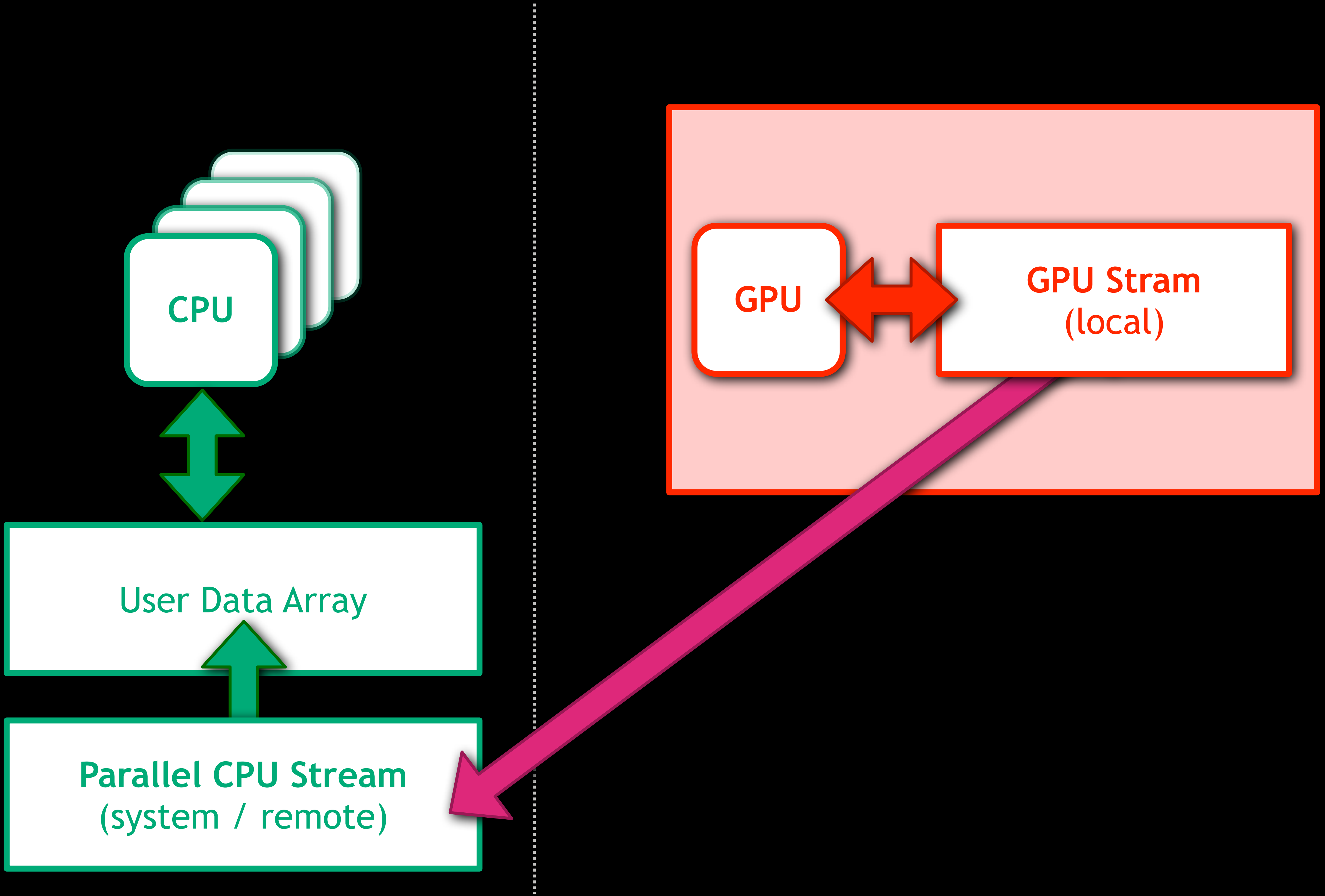
# streamWrite()



# streamWrite()



# streamWrite()



# Asynchronous stream transfers

Stream transfers are handled asynchronously by using the CAL asynchronous transfer mechanism

In Brook+, transfers are kicked off immediately and in the order of the stream calls

Transfers can occur in parallel with kernel executions not dependent on the streams being transferred.  Basically, transfers and kernel executions can be interleaved

Everything is handled by Brook+, so bottom line is just worry about coding your application

# Function passing

Brook+ supports stream function passing similar to C

Useful when creating larger apps

```
int foo(float in<>, int size)
{
    int val;
    ...
    return val;
}

int main(int argc, char** argv)
{
    int ret;
    float a<5>;

    ...
    ret = foo(a, 5);
    ...
}
```

# Stream domain modifier

The domain modifier allows sub-stream accesses with syntax:

*streamname*.domain(*start_address*, *end_address*);

*end_address* is not inclusive!

```
void printstream(float in<>, int size)
{
    int i;
    for(i=0; i<size; i++)
         ...
}

int main(int argc, char** argv)
{
     float my_a[5];
     float a1<10>;

     ...
     streamRead(a1.domain(2, 2+5), my_a);

     printstream(a1.domain(4, 7), 2);
     ...
}
```

# Stream domain modifier (cont.)

User is responsible for making sure dimensions match

Higher dimensional streams use vector addressing

Tip: What might be the problem with this code?

```
int main(int argc, char** argv)
{
    float my_b[2][3];
    float a2<10, 10>;
    int2 end;

    ...

    end = int2(3,5);
    streamWrite(a2.domain(int2(1,2), end), my_b);

    ...
}
```

# Stream domain modifier (cont.)

User is responsible for making sure dimensions match

Higher dimensional streams use vector addressing

Tip: What might be the problem with this code?

```
int main(int argc, char** argv)
{
    float my_b[2][3];
    float a2<10, 10>;
    int2 end;

    ...

    end = int2(3,5);
    streamWrite(a2.domain(int2(1,2), end), my_b);

    ...
}
```

**Watch the dimensions!**

# Kernels

# Brook+ kernels

Kernels are written like C functions with keyword “kernel”

Limitations

- All variables are automatic
- Pointers are not supported
- Memory cannot be allocated
- Recursion is not allowed
- See spec for more details

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

# Supported data types

Standard types

- float - 32-bit floating point
- double - 64-bit floating point

Other types are promoted to float in kernel

- int - 32-bit signed integer
- bool - Boolean

Structs are also supported in kernel

# Standard stream passing

Standard stream passing using open brackets - “<>”

Input and and output stream position is implicit

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

# Standard stream passing

Standard stream passing using open brackets - “<>”

Input and and output stream position is implicit

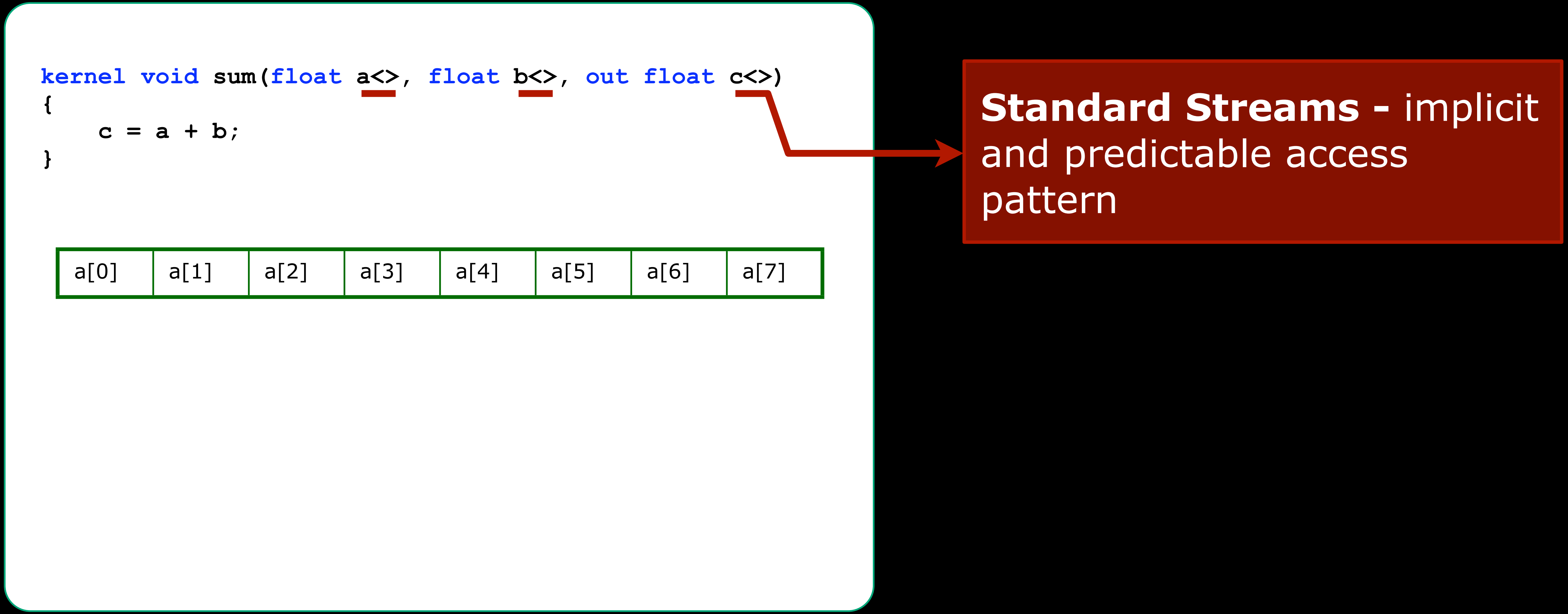
```
kernel void sum(float a<>, float b<>, out float c<>)
{
    c = a + b;
}
```

**Standard Streams -** implicit and predictable access pattern

# Standard stream passing

Standard stream passing using open brackets - “<>”

Input and and output stream position is implicit



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

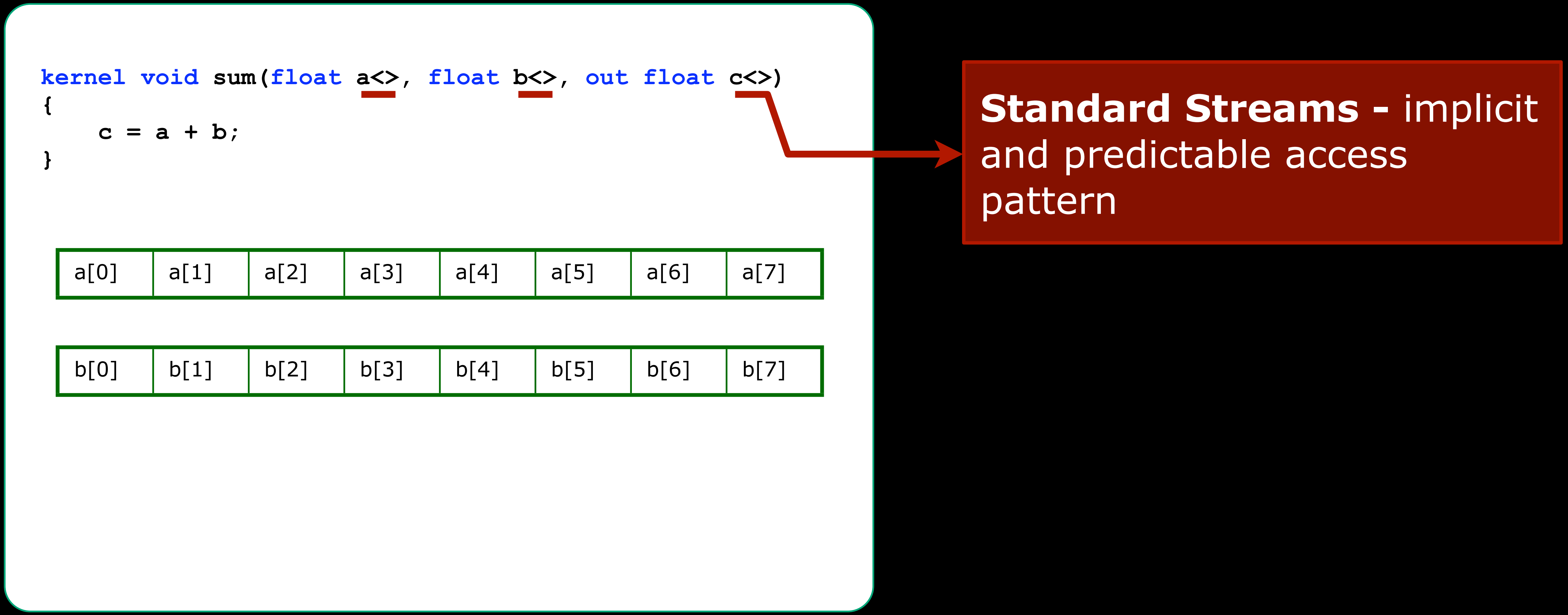
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |

**Standard Streams -** implicit and predictable access pattern

# Standard stream passing

Standard stream passing using open brackets - “<>”

Input and and output stream position is implicit



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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |

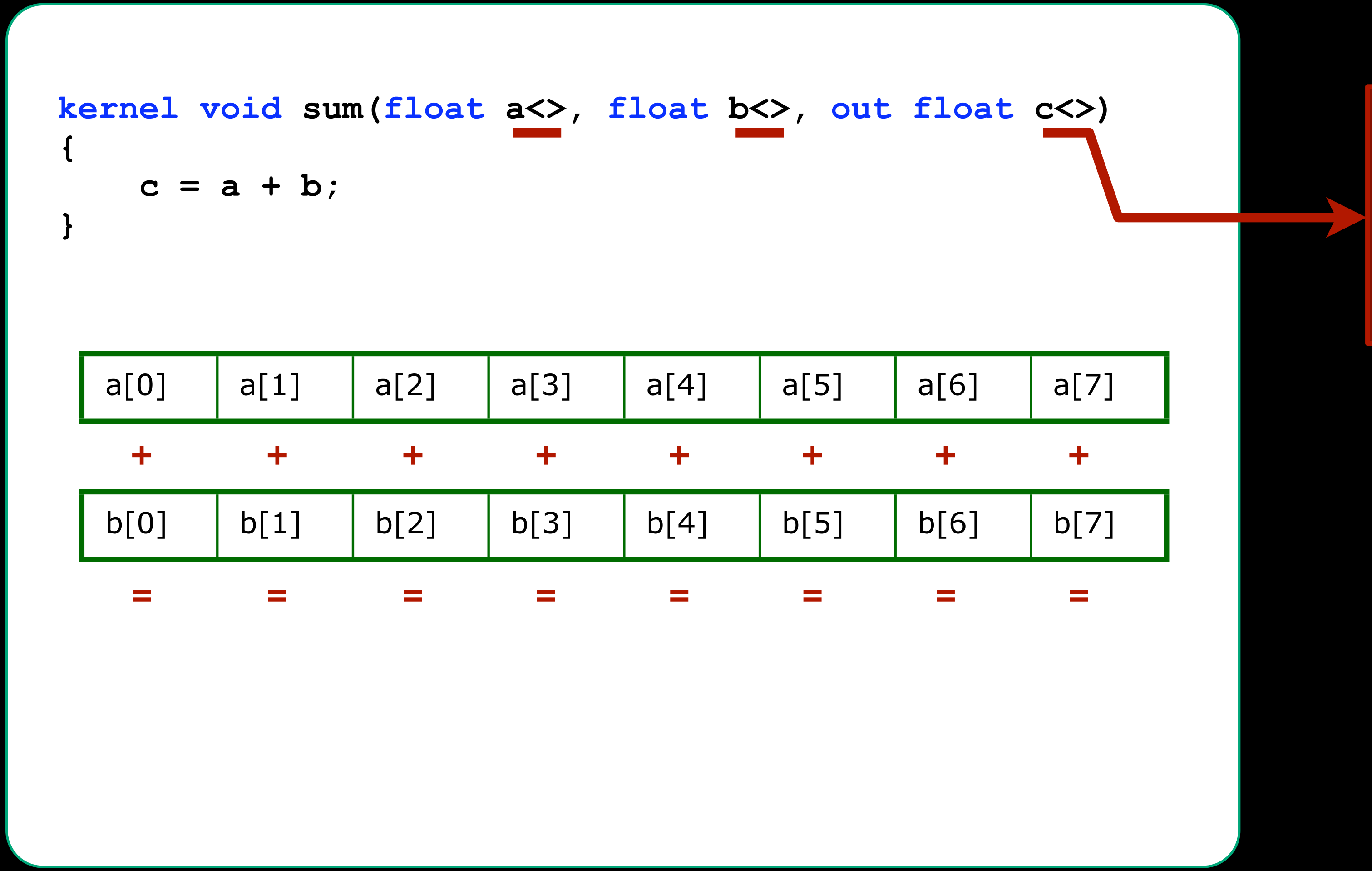
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| b[0] | b[1] | b[2] | b[3] | b[4] | b[5] | b[6] | b[7] |

**Standard Streams -** implicit and predictable access pattern

# Standard stream passing

Standard stream passing using open brackets - “<>”

Input and and output stream position is implicit



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

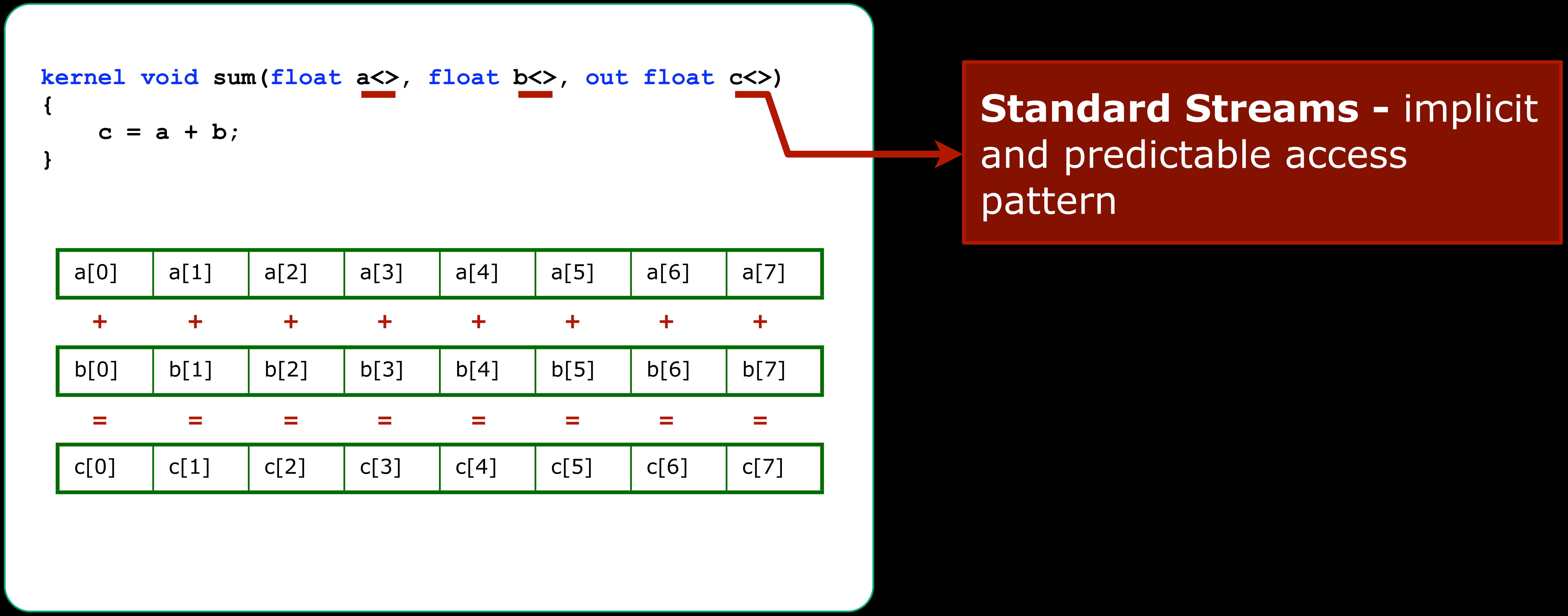
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |
| + | + | + | + | + | + | + | + |
| b[0] | b[1] | b[2] | b[3] | b[4] | b[5] | b[6] | b[7] |
| = | = | = | = | = | = | = | = |

**Standard Streams -** implicit and predictable access pattern

# Standard stream passing

Standard stream passing using open brackets - “<>”

Input and and output stream position is implicit



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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |
| + | + | + | + | + | + | + | + |
| b[0] | b[1] | b[2] | b[3] | b[4] | b[5] | b[6] | b[7] |
| = | = | = | = | = | = | = | = |
| c[0] | c[1] | c[2] | c[3] | c[4] | c[5] | c[6] | c[7] |

**Standard Streams -** implicit and predictable access pattern

# Standard stream passing

Standard stream passing using open brackets - “<>”

Input and and output stream position is implicit

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

**Standard Streams -** implicit and predictable access pattern

# Gather streams

Kernel stream input parameters declared with square brackets - “[]” - are considered gather streams

Gather streams can be arbitrarily addressed

indexof(*streamname*) function returns current position in the stream

```
kernel void sum(float a[], float b[], out float c<>)
{
     int idx = indexof(c);
     c = a[idx] + b[idx];
}
```

**Gather Streams -** dynamic read access pattern

# Scatter streams

Writing to arbitrary memory locations is known as *scatter* and is a relatively new feature of the GPU

Scatter output streams is declared with square brackets

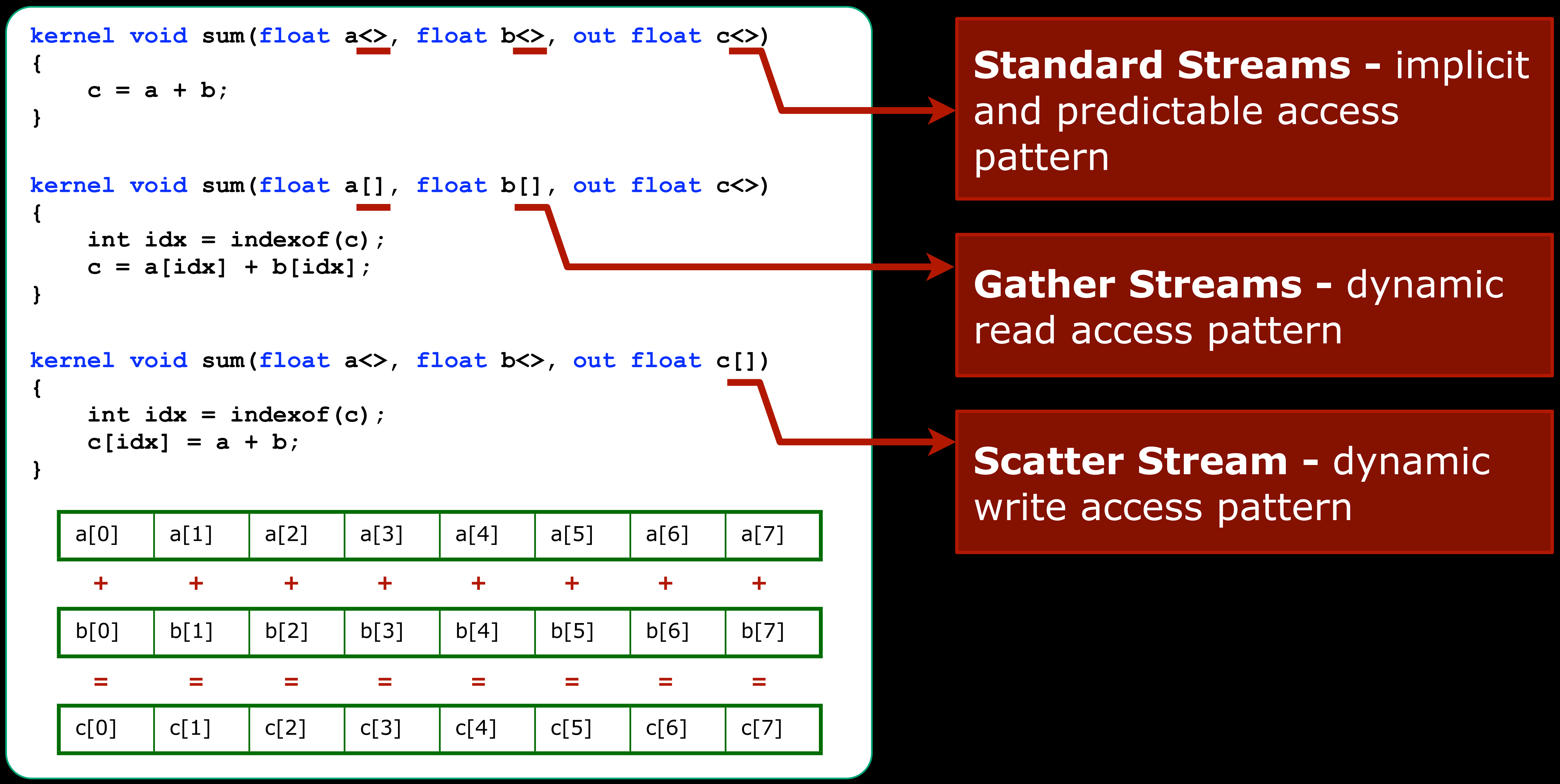
Calling indexof on a scatter stream has special meaning and will be discussed later

```
kernel void sum(float a<>, float b<>, out float c[])
{
     int idx = indexof(c);
     c[idx] = a + b;
}
```

**Scatter Stream -** dynamic write access pattern

# Equivalent kernels

These kernels, with parameter conventions, do the same thing



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

```
kernel void sum(float a[], float b[], out float c<>)
{
    int idx = indexof(c);
    c = a[idx] + b[idx];
}
```

```
kernel void sum(float a<>, float b<>, out float c[])
{
    int idx = indexof(c);
    c[idx] = a + b;
}
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |
| + | + | + | + | + | + | + | + |
| b[0] | b[1] | b[2] | b[3] | b[4] | b[5] | b[6] | b[7] |
| = | = | = | = | = | = | = | = |
| c[0] | c[1] | c[2] | c[3] | c[4] | c[5] | c[6] | c[7] |

**Standard Streams -** implicit and predictable access pattern

**Gather Streams -** dynamic read access pattern

**Scatter Stream -** dynamic write access pattern

# Input/Output HW Limitations

Both standard and gather streams can be mixed in a kernel for input and output

Up to 16 input streams of standard or gather type can be declared

Kernels can only write to 8 standard output streams

Only one output stream can be declared as scatter

# Constants

Constants can be passed from the application to the kernel

This can be useful for passing stream dimensions

```
kernel void sum(float a<>, float b<>, float2 offset, out float c<>)
{
    c = a + b + offset.x - offset.y;
}

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];

    ...

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

    sum(a, b, float2(5.f, 3.f), c);

    streamWrite(c, input_c);
    ...
}
```

# Calling other code from kernel code

Kernels can call other functions

Kernel functions must use keyword *kernel*

```
kernel float helper(float x, float y)
{
   return x + y;
}

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

# Reduction

Reduction collapses a stream along one axis using an associate, commutative binary operation (e.g. +=, *=)

Order of operations is not defined

For example, *find_min* finds the minimum value in a stream

```
reduce void find_min( float a<>, reduce float min )
{
    if( a < min ) min = a;
}
```

**Reduction -** programatically reduce stream to a value.

# Reduction (cont.)

Dimensional reductions are supported (i.e., 2D to 1D)

- float s<100, 200> reduced to float t<100>

Partial reductions are supported if sizes are integer multiples

- float s<100, 200> reduced to float t<100, 50>

A kernel may not generate both a reduced output and a conventional stream output

Multiple reduce variables are permitted

Reduce kernels do not have to produce an output for every input

AMD
Smarter Choice

# Kernel Execution

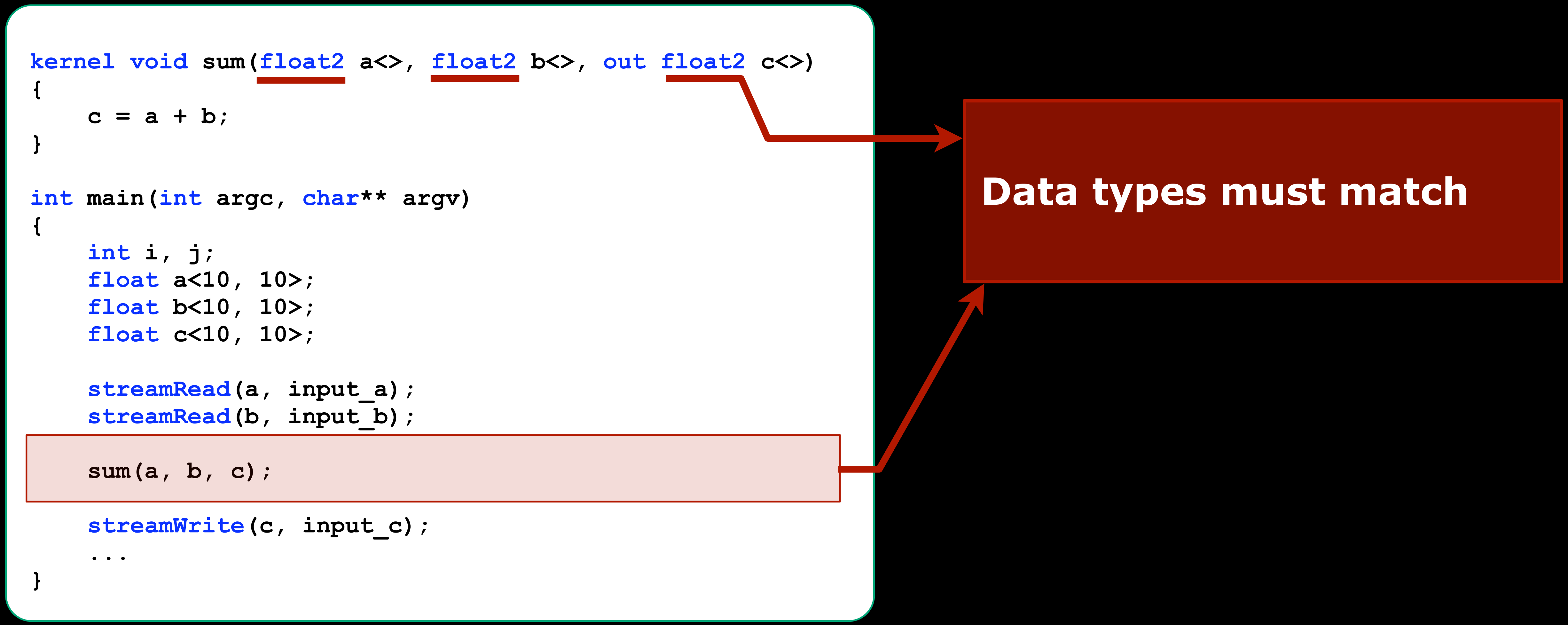
# Putting it all together

```
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);
    ...
}
```

**Kernels are called like C functions**

# Passing Streams

Data type must match between input streams in kernel parameters



```
kernel void sum(float2 a<>, float2 b<>, out float2 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>;

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

    sum(a, b, c);

    streamWrite(c, input_c);
    ...
}
```

**Data types must match**

# Stream domains

Kernels can be called with stream subdomains

Again, stream domains must match if using standard stream passing

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

int main(int argc, char** argv)
{
    float a<10>;
    float b<10>;
    float c<5>;

    ...

    sum(a.domain(2, 2+5), b.domain(5, 5+5), c;

    ...
}
```

# Interleaving

```
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 x<10, 10>;
   float y<10, 10>;
   float z<10, 10>;

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

   sum(a, b, c);

   streamRead(y, input_y);

   sum(x, y, z);

   streamWrite(z, input_z);
   streamWrite(c, input_c);
   ...
}
```

Synchronization is handled by the Brook+ runtime

GPU operations are non-blocking unless synchronization is needed

Be careful of ordering

**streamRead(x) will not block sum(a,b)**

**streamWrite(z) will block streamWrite(c)**

# Scatter

Only 1D scatter streams currently supported

*streamname*.execDomain(*domain_length*)

```
kernel void sum(float a[], float b[], out float c[])
{
    int idx = indexof(c);
    c[idx] = a[idx] + b[idx];
}

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

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

    sum(a, b, c.execDomain(10));

    streamWrite(c, input_c);
    ...
}
```

# Compiler optimizations

Brook+ compiler is really just a code generator. Very little optimizations are happening

IL kernels are optimized by the CAL driver compiler at runtime

C++ compiler isn't smart enough to remove dead GPU code (e.g. kernels with outputs that are never used)

# Bottom line…

Brook+ tries to be like C

Be smart, don’t force stuff that clearly doesn’t make GPU sense

Read the spec and programming guide

Refer to the included samples

Give us feedback!

# BREAK!

# Brook+ Development Environment

# Writing Brook+ applications

Brook+ consists of two parts

- –Brook+ compiler that generates C code
- –Brook+ runtime library that handles GPU calls

Brook+ code is written in Brook+ files usually with a ".br" extension

# Writing Brook+ applications

Brook+ code is written in Brook+ files usually with a ".br" extension

Brook+ files are compiled using the Brook+ compiler (brcc.exe), which generates C code along with GPU code (e.g. shaders, API calls)

Generated code is compiled to binaries with brook runtime libraries

Generated code could also be compiled with other application code.

C language is needed for code generation only.  Application code could be written in C++

# Example walkthrough

Environment variables

Code writing

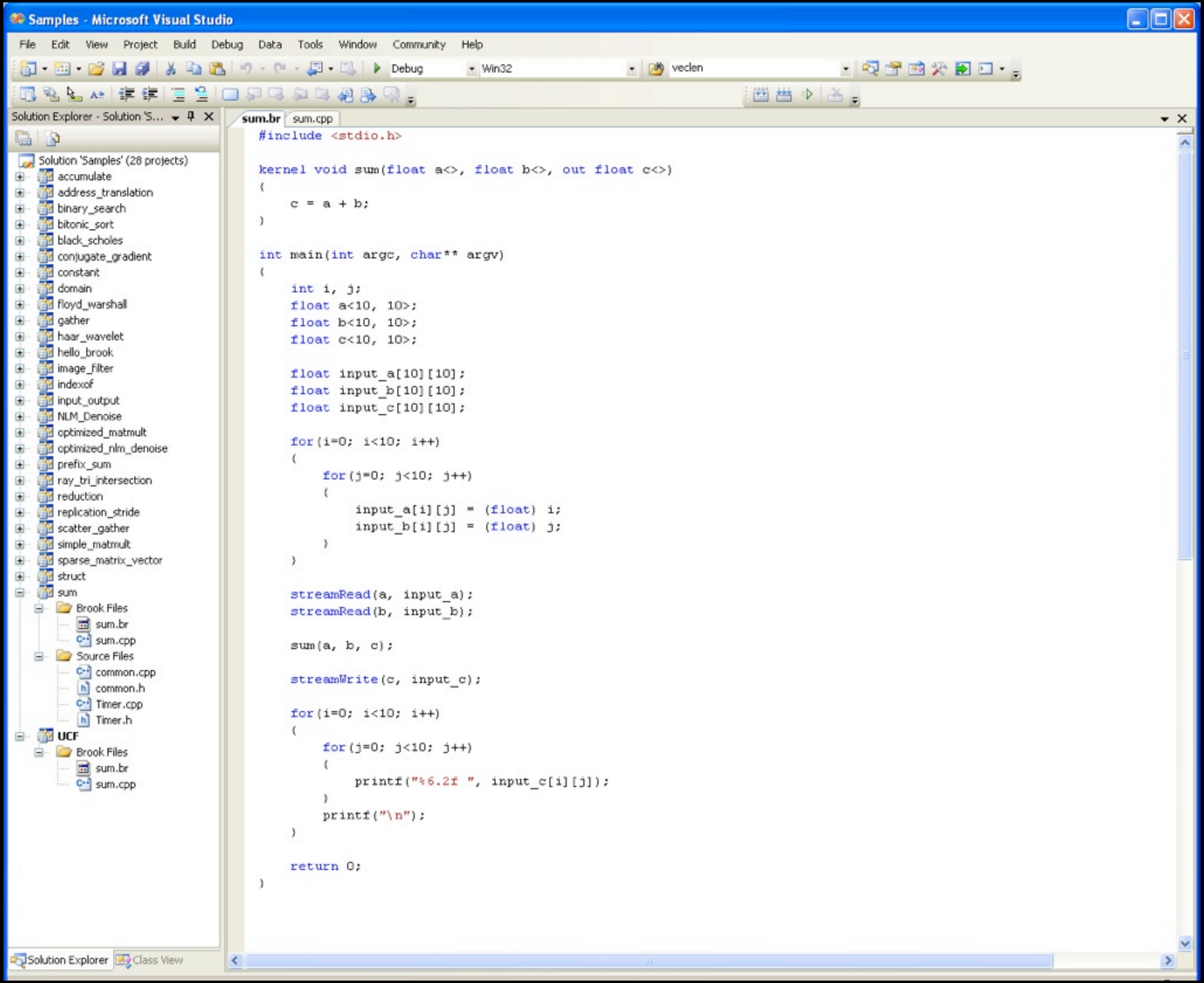
Code generation through brcc

Linking with runtime library

Debugging

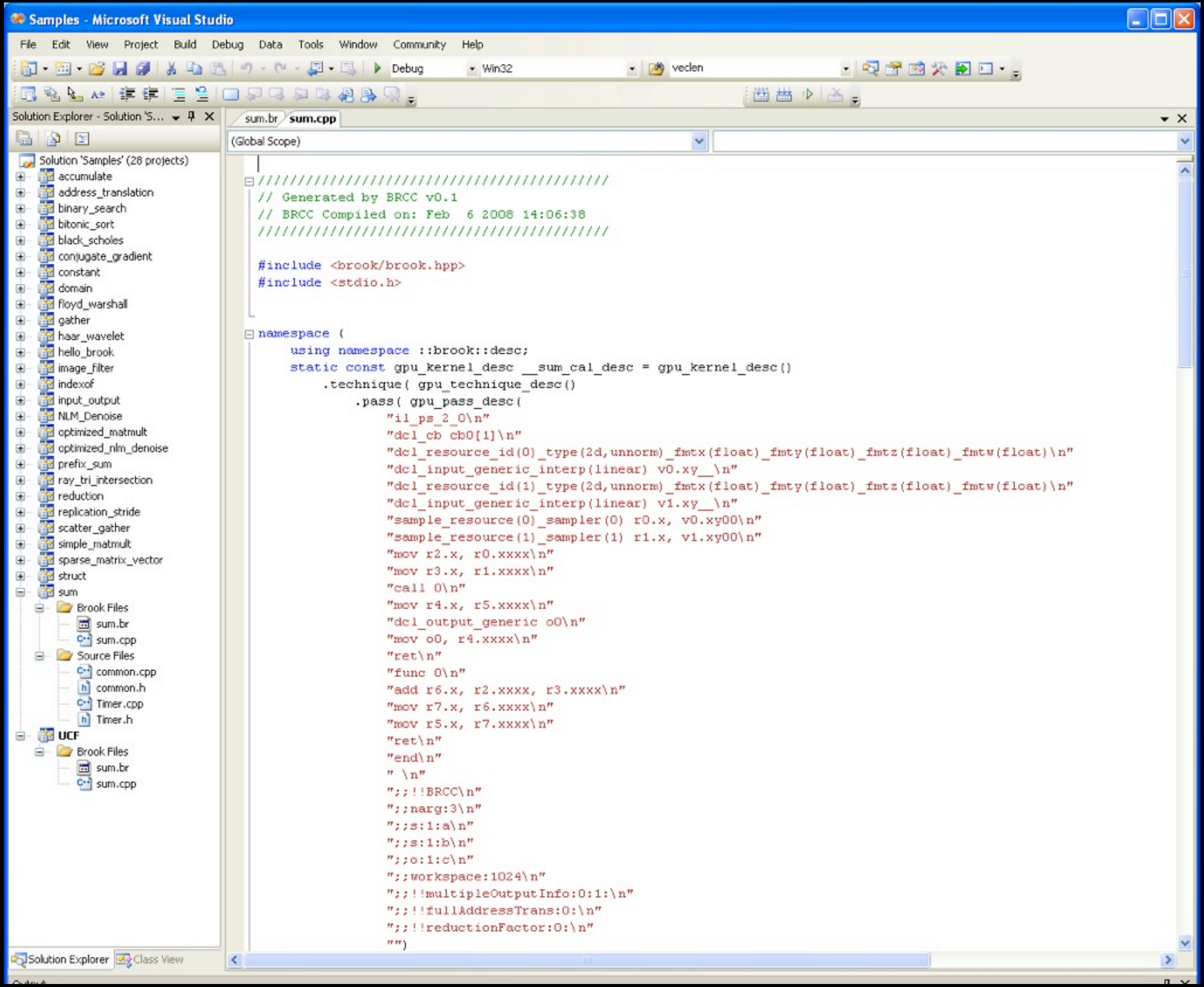
Samples

# Brook+ code



```
#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;
}
```

# Generated CPP code



```
////////////////////////////////////////////
// Generated by BRCC v0.1
// BRCC Compiled on: Feb  6 2008 14:06:38
////////////////////////////////////////////

#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"
                "")
```

# Debugging

"printf" debugging by outputting intermediate values to secondary buffers

Debug using the CPU backend

- –set environment variable "*brt_runtime=cpu*" (default is cal)

# Address Translation

Address translation is the ability to support large textures using software address calculations

Basically support larger sizes or dimensions

Address translation can have a performance penalty if used unnecessarily

Brook+ compiler will generate both types of code and automatically pick between the two

Use -R to compile only non-address translated code

# Creating larger applications

Use separate files

Treat .br files almost like a library

Keep Brook+ functions to a minimum if possible