Spatial Documentation

Release 0.1

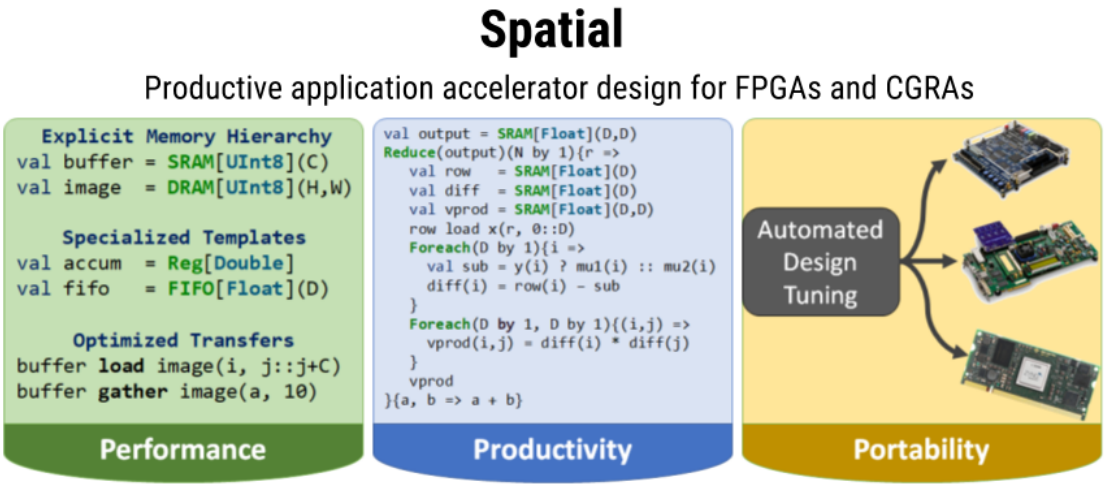
Stanford PPL

Feb 24, 2018

Contents

For a brief introduction to Spatial and its purpose, see this presentation.

For a brief rundown of the language and its constructs, see these slides and the *in-depth tutorials* on this website.



Get Started	*API Documentation*	Source

Spatial is a domain-specific language for describing hardware accelerators for use on FPGAs and other supported spatial architectures. The language is intended to be both higher level than hardware description languages (HDLs) like Verilog, VHDL, and Chisel, while also being easier to use than Altera's OpenCL or high level synthesis (HLS) languages like Xilinx's Vivado.

Language features of Spatial include:

- Tunable, hardware specific templates
- User specified and implicitly created design parameters
- Design runtime and area analysis
- Automatic parameter tuning
- Automatic memory banking and buffering

To run Spatial on the new Amazon EC2 FPGA instances, see *the AWS Tutorial*.

CHAPTER 1

API

The Spatial API is divided into three components: **Acceleratable** data structures and syntax can be used in the acceleration scope, while **Host** operations can only be used within the host code. Operations which are often used for either within the accelerator definition or in the host code are grouped under the **Common** category.

Documentation for running the Spatial compiler is provided under **Compiler**.

1.1 Common

This section lists operations and types which are available for use both on the CPU and in hardware *Accel* scopes.

1.1.1 Data Structures

This subsection lists data structures which are available for use both on the CPU and in hardware *Accel* scopes.

Range

Range represents a sequence of 32b integer values from a given *start* (inclusive) to *end* (exclusive) with a non-zero step size. Range also has an optional parallelization factor, which is used to determine physical unrolling in hardware.

In Accel scopes, Range instances can be implicitly converted to *Counter*.

Ranges can be created using two different syntax flavors. When unspecified, the default value for *start* is 0, and the default value for *step* and *par* are both 1.

Form 1:

```
val range1 = max by step        // start of 0, step by 1, parallelization of 1
val range2 = start until max    // start of 0, step by 1, parallelization of 1
val range3 = start until max by step par p
```

Form 2:

```
val range4 = start :: end       // Step of 1, parallelization of 1
val range5 = start :: step :: end par p
```

Form 1 is typically used for specifying the domain of a loop, while **Form 2** is used for Matlab-like syntax when selecting a range of elements from a memory.

Ranges can also be used to create Scala-like for loops on the CPU::

```
for (i <- 0 until max by 2) {
    ..
}
```

This loop will run for all even integers from [0, max). Note that this syntax is only usable on the CPU or in simulation.

Infix methods

class **Range**
def **length**: *Index* Returns the length of this Range.
def **by**(step: *Index*): *Range* Creates a Range with this Range's start and end but with the given **step** size.
def **par**(p: *Index*): *Range* Creates a Range with this Range's start, end, and stride, but with the given **par** parallelization factor.
def **::**(start2: *Index*): *Range* Creates a Range with this Range's end, with the previous start as the step size, and the given start. Note that this operator is right-associative.
def **foreach**(func: *Index* => *Unit*): *Unit* Iterates over all integers in this range, calling **func** on each. **NOTE**: This method is unsynthesizable, and can be used only on the CPU or in simulation.

Implicit methods

def **rangeToCounter**(range: *Range*): *Counter* Implicitly creates a hardware *Counter* from this Range.

Structs

Custom structs can be created in Spatial using the **@struct** macro.

For example:

```
@struct MyData(x: Int, y: Double)
```

creates a struct which contains an *Int* and a *Double*. An instance of this struct can be created using:

```
val data = MyData(32, 45.0f)
```

Note that this instance is immutable - its fields cannot be updated. Mutable custom struct instances are currently not supported, but may be added in the future.

The fields of a custom struct are accessed using:

```
data.x

data.y
```

Defining a custom struct also implicitly adds evidence of the *Bits* and *Arith* type classes. This allows syntax like:

```
val rand2 = random[MyData]

val rand1 = random[MyData](MyData(32, 2.0f))

val a = rand2 + rand1
val b = rand2 - rand1
```

Tuple2

Tuple2[A,B] is a simple data structure used to hold a pair of staged values.

Note that this name shadows the unstaged Scala type. For the unstaged type, use the full name scala.Tuple2[A,B]

Infix methods

class **Tuple2**[A,B]
def **_1**: A Returns the first field in this Tuple2.
def **_2**: B Returns the second field in this Tuple2.
def **toString**: *String* Returns a printable String from this value. **NOTE**: This method is unsynthesizable and can only be used on the CPU or in simulation.
def **———-**
def ****Related** methods**
def **@table-start**
def **NoHeading**
def **pack**[A:*Type*,B:*Type*](t: (A, B)): MTuple2[A,B] Returns a staged Tuple2 from the given unstaged Tuple2.
def **pack**[A:*Type*,B:*Type*](a: A, b: B): MTuple2[A,B] Returns a staged Tuple2 from the given pair of values. Shorthand for ****pack((a,b))****.
def **unpack**[A:*Type*,B:*Type*](t: MTuple2[A,B]): (A,B) Returns an unstaged scala.Tuple2 from this staged Tuple2. Shorthand for ****(x._1, x._2)****.

Vector

Vector defines a fixed size collection of scalar values. It is distinct from *Array* in that the size must always be statically determinable. This allows Vector to always be allocated as a bus of wires when implemented in hardware.

Vector is most commonly used for managing parallel accesses to local memories in hardware and for bit-twiddling operations.

To allow static type checking and creation of the *Bits* type class, the Vector type is split into subtypes based on the number of elements it contains. For example, a Vector3[*Int*] is composed of 3, 32-bit *Int* values, while a Vector32[*Bit*] is a single bus of 32 *Bit* values. Spatial currently defines Vector types from **Vector1** up to **Vector128**.

To work around *limitations with Scala's type system* Spatial also includes a VectorN type for when the vector width cannot be type-encoded. This type generally needs to be annotated with the number of bits the user intended (e.g. `vector.as32b`) before it can be written to memories.

Static methods

object **Vector**
def **LittleEndian**[T:*Type*:*Bits*](elem: T*): *VectorX*[T] Creates a VectorX from the given X elements, where X is between 1 and 128. The first element supplied is the most significant (Vector index of X - 1). The last element supplied is the least significant (Vector index of 0). Note that this method is actually overloaded 128 times based on the number of supplied arguments.
def **BigEndian**[T:*Type*:*Bits*](elem: T*): *VectorX*[T] Creates a VectorX from the given X elements, where X is between 1 and 128. The first element supplied is the least significant (Vector index of 0). The last element supplied is the most significant (Vector index of X - 1). Note that this method is actually overloaded 128 times based on the number of supplied arguments.
def **ZeroLast**[T:*Type*:*Bits*](elem: T*): *VectorX*[T] A mnemonic for LittleEndian (with reference to the zeroth element being specified last in order).
def **ZeroFirst**[T:*Type*:*Bits*](elem: T*): *VectorX*[T] A mnemonic for BigEndian (with reference to the zeroth element being specified first in order).

Spatial also includes an alternate **Vectorize** object which takes a true arbitrary number of elements in all of its functions. As a result, these methods return VectorNs.

object **Vectorize**
def **LittleEndian**[T:*Type*:*Bits*](elem: T*): *VectorN*[T] Creates a VectorN from the given elements. The first element supplied is the most significant (Vector index of N - 1). The last element supplied is the least significant (Vector index of 0).
def **BigEndian**[T:*Type*:*Bits*](elem: T*): *VectorN*[T] Creates a VectorN from the given elements. The first element supplied is the least significant (Vector index of 0). The last element supplied is the most significant (Vector index of N - 1).
def **ZeroLast**[T:*Type*:*Bits*](elem: T*): *VectorN*[T] A mnemonic for LittleEndian (with reference to the zeroth element being specified last in order).
def **ZeroFirst**[T:*Type*:*Bits*](elem: T*): *VectorN*[T] A mnemonic for BigEndian (with reference to the zeroth element being specified first in order).

Infix methods

class **Vector**[T]

def **apply**(i: *Int*): T

Returns the **i**'th element of this Vector.
Element 0 is always the LSB.

def **apply**(range: *Range*)(implicit mT: *Type*[T], bT: *Bits*[T]): *VectorN*[T]

Returns a slice of the elements in this Vector as a VectorN.
The range must be statically determinable with a stride of 1.
The range is inclusive for both the start and end.
The **range** can be big endian (e.g. **3::0**) or little endian (e.g. **0::3**).
In both cases, element 0 is always the least significant element.

For example, **x(3::0)** returns a Vector of the 4 least significant elements of **x**.

def **takeX**(offset: scala.Int): *VectorX*[T]

Returns a slice of N elements of this Vector starting at the given **offset** from the
least significant element.
To satisfy Scala's static type analysis, each width has a separate method.

For example, **x.take3(1)** returns the 3 least significant elements of x after the
least significant as a Vector3[T].

def **!=**(that: *Vector*[T]): MBoolean

Returns true if this Vector and **that** differ by at least one element, false otherwise.

def **==**(that: *Vector*[T]): MBoolean

Returns true if this Vector and **that** contain the same elements, false otherwise.

class **VectorX**[T] extends Vector[T]

class **VectorN**[T]

def **asVectorX**: *VectorX*[T]

Casts this VectorN as a VectorX.
Values of X from 1 to 128 are currently supported.

If the VectorX type has fewer elements than this value's type, the most significant elements will be dropped.
If the VectorX type has more elements than this value's type, the resulting elements will be zeros.

def **asXb**: *VectorX*[*Bit*]

Returns a view of this VectorN's bits as a X-bit Vector.
To satisfy Scala's static analysis, each bit-width has a separate method.
Conversions between 1 and 128 bits are currently supported.

If X is smaller than this VectorN's total bits, the MSBs will be dropped.
If X is larger than this VectorN's total bits, the resulting MSBs will be zeros.

1.1.2 Primitive Types

Boolean

Boolean represents a staged single boolean value.

Note that this type shadows the unstaged Scala Boolean. In the case where an unstaged Boolean type is required, use the full *scala.Boolean* name.

Infix methods

class **Boolean**
def **unary_!**: *Boolean* Negates the given boolean expression.
def **&&**(y: *Boolean*): *Boolean* Boolean AND. Compares two Booleans and returns *true* if both are *true*.
def **||**(y: *Boolean*): *Boolean* Boolean OR. Compares two Booleans and returns *true* if at least one is *true*.
def **^**(y: *Boolean*): *Boolean* Boolean exclusive-or (XOR). Compares two Booleans and returns *true* if exactly one is *true*.
def **!=**(y: *Boolean*): *Boolean* Value inequality (equivalent to exclusive-or). Compares two Booleans and returns *true* if exactly one is *true*.
def **==**(y: *Boolean*): *Boolean* Value equality, equivalent to exclusive-nor (XNOR). Compares two Booleans and returns *true* if both are *true* or both are *false*
def **toString**: *String* Creates a printable String from this value [**NOTE**] This method is unsynthesizable, and can be used only on the CPU or in simulation.

FixPt

FixPt[S,I,F] represents an arbitrary precision fixed point representation. FixPt values may be signed or unsigned. Negative values, if applicable, are represented in twos complement.

The type parameters for FixPt are:

S	BOOL	Signed representation	TRUE | FALSE
I	INT	Number of integer bits	(_1 - _64)
F	INT | Number of fractional bits		(_0 - _64)

Note that numbers of bits use the underscore prefix as integers cannot be used as type parameters in Scala.

Type Aliases

Specific types of FixPt values can be managed using type aliases. New type aliases can be created using syntax like the following:

```
type Q16_16 = FixPt[TRUE,_16,_16]
```

Spatial defines the following type aliases by default:

type	IntN	*FixPt*[TRUE,_N,_0]	Signed, N bit integer (_2 - _128)
type	UIntN	*FixPt*[TRUE,_N,_0]	Unsigned, N bit integer (_2 - _128)
type	Char	*FixPt*[TRUE,_8,_0]	Signed, 8 bit integer
type	Short	*FixPt*[TRUE,_16,_0]	Signed, 16 bit integer
type	Int	*FixPt*[TRUE,_32,_0]	Signed, 32 bit integer
type	Index	*FixPt*[TRUE,_32,_0]	Signed, 32 bit integer (indexing)
type	Long	*FixPt*[TRUE,_64,_0]	Signed, 64 bit integer

Note that the Char, Short, Int, and Long types shadow their respective unstaged Scala types. In the case where an unstaged type is required, use the full *scala.** name.

Infix methods

The following infix methods are defined on all FixPt classes. When the method takes a right hand argument, only values of the same FixPt class can be used for this argument.

class **FixPt**[S,I,F]
def **unary_-**(): *FixPt*[S,I,F] Returns negation of this fixed point value.
def **unary_~**(): *FixPt*[S,I,F] Returns bitwise inversion of this fixed point value.
def **+** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Fixed point addition.
def **-** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Fixed point subtraction.
def ***** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Fixed point multiplication.
def **/** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Fixed point division.
def **%** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Fixed point modulus.

Continued on next page

Table 1.1 – continued from previous page

class **FixPt**[S,I,F]
def **(exp: scala.Int): *FixPt*[S,I,F] Integer exponentiation, implemented in hardware as a reduction tree with **exp** inputs.
def ***&** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Fixed point multiplication with unbiased rounding. After multiplication, probabilistically rounds up or down to the closest representable number.
def **/&** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Fixed point division with unbiased rounding. After division, probabilistically rounds up or down to the closest representable number.
def **<+>** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Saturating fixed point addition. Addition which saturates at the largest or smallest representable number upon over/underflow.
def **<->** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Saturating fixed point subtraction. Subtraction which saturates at the largest or smallest representable number upon over/underflow.
def **<*>** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Saturating fixed point multiplication. Multiplication which saturates at the largest or smallest representable number upon over/underflow.
def **</>** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Saturating fixed point division. Division which saturates at the largest or smallest representable number upon over/underflow.
def **<*&>** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Saturating fixed point multiplication with unbiased rounding. After multiplication, probabilistically rounds up or down to the closest representable number. After rounding, also saturates at the largest or smallest representable number upon over/underflow.
def **</&>** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Saturating fixed point division with unbiased rounding. After division, probabilistically rounds up or down to the closest representable number. After rounding, also saturates at the largest or smallest representable number upon over/underflow.

Continued on next page

Table 1.1 – continued from previous page

class **FixPt**[S,I,F]
def **<** (that: *FixPt*[S,I,F]): MBoolean Less than comparison. Returns **true** if this value is less than **that** value. Otherwise returns **false**.
def **<=**(that: *FixPt*[S,I,F]): MBoolean Less than or equal comparison. Returns **true** if this value is less than or equal to **that** value. Otherwise returns **false**.
def **>** (that: *FixPt*[S,I,F]): MBoolean Greater than comparison Returns **true** if this value is greater than **that** value. Otherwise returns **false**.
def **>=**(that: *FixPt*[S,I,F]): MBoolean Greater than or equal comparison. Returns **true** if this value is greater than or equal to **that** value. Otherwise returns **false**.
def **!=**(that: *FixPt*[S,I,F]): *Boolean* Value inequality comparison. Returns **true** if this value is not equal to the right hand side. Otherwise returns **false**.
def **!=**(that: *FixPt*[S,I,F]): *Boolean* Value equality comparison. Returns **true** if this value is equal to the right hand side. Otherwise returns **false**.
def **&** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Bit-wise AND.
def **|** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Bit-wise OR.
def **^** (that: *FixPt*[S,I,F]): *FixPt*[S,I,F] Bit-wise XOR.
def **<<**(that: *FixPt*[S,I,_0]): *FixPt*[S,I,F] Logical shift left.
def **>>**(that: *FixPt*[S,I,_0]): *FixPt*[S,I,F] Arithmetic (sign-preserving) shift right.
def **>>>**(that: *FixPt*[S,I,_0]): *FixPt*[S,I,F] Logical (zero-padded) shift right.

Continued on next page

Table 1.1 – continued from previous page

class **FixPt**[S,I,F]
def **as**[T:*Type*:*Bits*]: T Re-interprets this value's bits as the given type, without conversion.
def **apply**(i: scala.Int): *Bit* Returns the given bit in this value. 0 corresponds to the least significant bit (LSB).
def **apply**(range: *Range*): *Vector*[*Bit*] Returns a vector of bits based on the given range. The range must be statically determinable values.
def **reverse**: *FixPt*[S,I,F] Returns a fixed point value with this value's bits in reverse order.
def **to**[T:*Type*:*Bits*]: T Converts this value to the given type. Currently supported types are *FixPt*, *FltPt*, and *String*.
def **toString**: *String* Creates a printable String representation of this value. **NOTE**: This method is unsynthesizable, and can be used only on the CPU or in simulation.

Specialized infix methods

These methods are defined on only specific classes of FixPt values.

subclass **Int** (aliases: **Index**, **FixPt**[TRUE, _32, _0])
def **::**(end: *Int*): *Range* Creates a Range with this as the start (inclusive), the given end (noninclusive), and step of 1.
def **by**(step: *Int*): *Range* Creates a Range with start of 0 (inclusive), this value as the end (noninclusive), and the given step.
def **until**(end: *Int*): *Range* Creates a Range with this as the start (inclusive), the given end (noninclusive), and step of 1.

FltPt

FltPt[G,E] represents an arbitrary precision, IEEE-754-like representation. FltPt values are always assumed to be signed.

The type parameters for FltPt are:

G	INT	Number of significand bits, including sign bit	(_2 - _64)
E	INT	Number of exponent bits	(_1 - _64)

Note that numbers of bits use the underscore prefix as integers cannot be used as type parameters in Scala.

Type Aliases

Specific types of FltPt values can be managed using type aliases. New type aliases can be created using::

```
type MyType = FltPt[_##,_##]
```

Spatial defines the following type aliases by default:

type	Half	*FltPt*[_11,_5]	IEEE-754 half precision
type	Float	*FltPt*[_24,_8]	IEEE-754 single precision
type	Double	*FltPt*[_53,_11]	IEEE-754 double precision

Note that the Float and Double types shadow their respective unstaged Scala types. In the case where an unstaged type is required, use the full *scala.** name.

Infix methods

class **FltPt**[G,E]

def **unary_-**(): *FltPt*[G,E]
Returns the negation of this floating point value.

def **+** (that: *FltPt*[G,E]): *FltPt*[G,E]
Floating point addition.

def **-** (that: *FltPt*[G,E]): *FltPt*[G,E]
Floating point subtraction.

def * (that: *FltPt*[G,E]): *FltPt*[G,E]
Floating point multiplication.

def **/** (that: *FltPt*[G,E]): *FltPt*[G,E]
Floating point division.

def **(exp: scala.Int): *FltPt*[G,E]
Integer exponentiation, implemented in hardware as a reduction tree with **exp** inputs.

def **<** (that: *FltPt*[G,E]): MBoolean
Less than comparison.

Returns **true** if this value is less than **that** value. Otherwise returns **false**.

def **<=**(that: *FltPt*[G,E]): MBoolean
Less than or equal comparison.

Returns **true** if this value is less than or equal to **that** value. Otherwise returns **false**.

def **>** (that: *FltPt*[G,E]): MBoolean
Greater than comparison.

Returns **true** if this value is greater than **that** value. Otherwise returns **false**.

def **>=**(that: *FltPt*[G,E]): MBoolean
Greater than or equal comparison.

Returns **true** if this value is less than **that** value. Otherwise returns **false**.

def **!=**(that: *FltPt*[G,E]): *Boolean*
Value inequality comparison.
Returns **true** if this value is not equal to the right hand side. Otherwise returns **false**.

def **!=**(that: *FltPt*[G,E]): *Boolean*
Value equality comparison.
Returns **true** if this value is equal to the right hand side. Otherwise returns **false**.

def **as**[T:*Type*:*Bits*]: T
Re-interprets this value's bits as the given type, without conversion.

def **apply**(i: scala.Int): *Bit*
Returns the given bit in this value.
0 corresponds to the least significant bit (LSB).

def **apply**(range: *Range*): *Vector*[*Bit*]

Unit

Like Scala, **Unit** is a type representing no return value in Spatial. It is similar to *void* in Java or C++.

1.1.3 Type Classes

Arith

Type class used to supply evidence that type T has basic arithmetic operations defined on it.

Abstract methods

trait **Arith**[T]
def **negate**(x: T): T Returns the negation of the given value.
def **plus**(x: T, y: T): T Returns the result of adding **x** and **y**.
def **minus**(x: T, y: T): T Returns the result of subtracting **y** from **x**.
def **times**(x: T, y: T): T Returns the result of multiplying **x** and **y**.
def **divide**(x: T, y: T): T Returns the result of dividing **x** by **y**.

Bits

The Bits type class is used to supply evidence that a type T is representable by a statically known number of bits.

Abstract Methods

trait **Bits**[T]
def **length**: scala.Int Returns the minimum number of bits required to represent type T.
def **zero**: T Returns the zero value for type T.
def **one**: T Returns the one value for type T.
def **random**(max: Option[T]): T Generates a pseudorandom value uniformly distributed between 0 and max. If max is unspecified, type T's default maximum is used instead. For *FixPt* types, the default maximum is the maximum representable number. For *FltPt* types, the default maximum is 1. For composite *Tuple2* and *Struct* types, the maximum is determined per component.

Related methods

def **zero**[T:*Bits*]: T Returns the zero value for type T.
def **one**[T:*Bits*]: T Returns the one value for type T.
def **random**[T:*Bits*]: T Generates a pseudorandom value uniformly distributed between 0 and the default maximum for type T.
def **random**[T:*Bits*](max: T): T Generates a pseudorandom value uniformly distributed between 0 and **max**.

Infix methods

Instances of types which have evidence of Bits also have infix methods defined on them:

def **apply**(i: *Int*): *Bit*

Returns the given bit in this value.
0 corresponds to the least significant bit (LSB).

def **apply**(range: *Range*): BitVector

Returns a slice of the bits in this word as a VectorN.
The range must be statically determinable with a stride of 1.
The range is inclusive for both the start and end.
The range can be big endian (e.g. ****3::0****) or little endian (e.g. ****0::3****).
In both cases, element 0 is always the least significant element.

For example, ****x(3::0)**** returns a Vector of the 4 least significant bits of ****x****.

def **as**[B:*Type*:*Bits*]: B

Re-interprets this value's bits as the given type, without conversion.
If B has fewer bits than this value's type, the MSBs will be dropped.
If B has more bits than this value's type, the resulting MSBs will be zeros.

def **reverse**: A

Returns a value of the same type with this value's bits in reverse order.

def **takeX**(offset: scala.Int): *VectorX*[*Bit*]

Returns a slice of X bits of this value starting at the given **offset** from the LSB.
To satisfy Scala's static type analysis, each bit-width has a separate method.

For example, ****x.take3(1)**** returns the 3 least significant bits of x after the LSB
as a Vector3[Bit].

def **takeXMSB**(scala.Int): *VectorX*[*Bit*]

Returns a slice of X bits of this value, starting at the given **offset** from the MSB.
To satisfy Scala's static type analysis, each bit-width has a separate method.
Slices between 1 and 128 bits are currently supported.

For example, ****x.take3MSB(1)**** returns the 3 most significant bits of x after the MSB
as a Vector3[Bit].

def **asXb**: *VectorX*[*Bit*]

Returns a view of this value's bits as a X-bit Vector.
To satisfy Scala's static analysis, each bit-width has a separate method.
Conversions between 1 and 128 bits are currently supported.

If X is smaller than this value's total bits, the MSBs will be dropped.
If X is larger than this value's total bits, the resulting MSBs will be zeros.

Mem

Type class used to supply evidence that type T is a local memory, potentially with multiple dimensions.

Abstract Methods

trait **Mem**[T,C]
def **load**(mem: C[T], indices: Seq[*Index*], en: *Bit*): T Loads an element from **mem** at the given multi-dimensional address **indices** with enable **en**.
def **store**(mem: C[T], indices: Seq[*Index*], data: T, en: *Bit*): *Unit* Stores **data** into **mem** at the given multi-dimensional address **indices** with enable **en**.
def **iterators**(mem: C[T]): Seq[*Counter*] Returns a **Seq** of counters which define the iteration space of the given memory **mem**.
def **par**(mem: C[T]): Option[*Index*] Returns the parallelization annotation for this memory.

Num

Combination of *Arith*, *Bits*, and *Order* type classes

Abstract Methods

trait **Num**[T] extends Arith[T] with Bits[T] with Order[T]
def **toFixPt**[S:BOOL,I:INT,F:INT](x: T): *FixPt*[S,I,F] Converts **x** to a *FixPt* value.
def **toFltPt**[G:INT,E:INT](x: T): *FltPt*[G,E] Converts **x** to a *FltPt* value.
def **fromInt**(x: scala.Int, force: *Boolean* = true): T Returns the closest representable value of type T to **x**. If **force** is true, gives a warning when the conversion is not exact.
def **fromLong**(x: scala.Long, force: *Boolean* = true): T Returns the closest representable value of type T to **x**. If **force** is true, gives a warning when the conversion is not exact.
def **fromFloat**(x: scala.Float, force: *Boolean* = true): T Returns the closest representable value of type T to **x**. If **force** is true, gives a warning when the conversion is not exact.
def **fromDouble**(x: scala.Double, force: *Boolean* = true): T Returns the closest representable value of type T to **x**. If **force** is true, gives a warning when the conversion is not exact.
def **maxValue**: T Returns the largest, finite, positive value representable by type T.
def **minValue**: T Returns the most negative, finite value representable by type T.
def **minPositiveValue**: T Returns the smallest positive (nonzero) value representable by type T.

Order

Type class used to supply evidence that type T has basic ordering operations defined on it.

Abstract Methods

trait **Order**[T]
def **lessThan**(a: T, b: T): MBoolean Returns true if **a** is less than **b**, false otherwise.
def **lessThanOrEqual**(a: T, b: T): MBoolean Returns true if **a** is less than or equal to **b**, false otherwise.
def **equal**(a: T, b: T): MBoolean Returns true if **a** is equal to **b**, false otherwise.

Type

Type class used to supply evidence that type T is representable in the Spatial compiler.

This type class is primarily used to distinguish between unstaged values, i.e. Scala collections and primitives, and staged, Spatial values.

1.1.4 Math

Commonly used mathematical operators

Methods

def **abs**[T:*Type*:*Num*](value: T): T
Returns the absolute value of the supplied numeric **value**.

def **ceil**[S:BOOL,I:INT,F:INT](x: *FixPt*[S,I,F]): *FixPt*[S,I,F]
Returns the smallest (closest to negative infinity) integer value greater than or equal to **x**.

def **exp**[T:*Type*:*Num*](x: T)(implicit ctx: SrcCtx): T
Returns the natural exponentiation of **x** (e raised to the exponent **x**).

def **floor**[S:BOOL,I:INT,F:INT](x: *FixPt*[S,I,F]): *FixPt*[S,I,F]
Returns the largest (closest to positive infinity) integer value less than or equal to **x**.

def **log**[G:INT,E:INT](x: *FltPt*[G,E]): *FltPt*[G,E]
Returns the natural logarithm of **x**.

def **mux**[T:*Type*:*Bits*](select: *Bit*, a: T, b: T): T
Creates a multiplexer that returns **a** when **select** is true, **b** otherwise.

def **min**[T:*Type*:*Bits*:*Order*](a: T, b: T): T
Returns the minimum of the numeric values **a** and **b**.

def **max**[T:*Type*:*Bits*:*Order*](a: T, b: T): T
Returns the maximum of the numeric values **a** and **b**.

def **pow**[G:INT,E:INT](base: *FltPt*[G,E], exp:*FltPt*[G,E]): *FltPt*[G,E]
Returns **base** raised to the power of **exp**.

def **sqrt**[G:INT,E:INT](x: *FltPt*[G,E]): *FltPt*[G,E]
Returns the square root of **x**.

def **sin**[G:INT,E:INT](x: *FltPt*[G,E]): *FltPt*[G,E]
Returns the trigonometric sine of **x**.

def **cos**[G:INT,E:INT](x: *FltPt*[G,E]): *FltPt*[G,E]
Returns the trigonometric cosine of **x**.

def **tan**[G:INT,E:INT](x: *FltPt*[G,E]): *FltPt*[G,E]
Returns the trigonometric tangent of **x**.

def **sinh**[G:INT,E:INT](x: *FltPt*[G,E]): *FltPt*[G,E]
Returns the hyperbolic sine of **x**.

def **cosh**[G:INT,E:INT](x: *FltPt*[G,E]): *FltPt*[G,E]
Returns the hyperbolic cosine of **x**.

def **tanh**[G:INT,E:INT](x: *FltPt*[G,E]): *FltPt*[G,E]
Returns the hyperbolic tangent of **x**.

def **asin**[G:INT,E:INT](x: *FltPt*[G,E]): *FltPt*[G,E]
Returns the arc sine of **x**.

def **acos**[G:INT,E:INT](x: *FltPt*[G,E]): *FltPt*[G,E]
Returns the arc cosine of **x**.

def **atan**[G:INT,E:INT](x: *FltPt*[G,E]): *FltPt*[G,E]
Returns the arc tangent of **x**.

Approximate Methods

Some of the trigonometric functions are not yet supported in the Chisel backend yet. As a placeholder, or to minimize future hardware costs, the following Taylor series approximations are predefined:

def **sin_taylor**[S:BOOL,I:INT,F:INT](x: *FixPt*[S,I,F]): *FixPt*[S,I,F] Taylor series expansion for trigonometric sine from -pi to pi
def **cos_taylor**[S:BOOL,I:INT,F:INT](x: *FixPt*[S,I,F]): *FixPt*[S,I,F] Taylor series expansion for trigonometric cosine from -pi to pi
def **exp_taylor**[S:BOOL,I:INT,F:INT](x: *FixPt*[S,I,F]): *FixPt*[S,I,F] Taylor series expansion for natural exponential
def **exp_taylor**[G:INT,E:INT](x: *FltPt*[G,E]): *FltPt*[G,E] Taylor series expansion for natural exponential
def **log_taylor**[S:BOOL,I:INT,F:INT](x: *FixPt*[S,I,F]): *FixPt*[S,I,F] Taylor series expansion for natural log to third degree.
def **log_taylor**[G:INT,E:INT](x: *FltPt*[G,E]): *FltPt*[G,E] Taylor series expansion for natural log to third degree.
def **sqrt_approx**[S:BOOL,I:INT,F:INT](x: *FixPt*[S,I,F]): *FixPt*[S,I,F] Taylor series expansion for square root to third degree.
def **sqrt_approx**[G:INT,E:INT](x: *FltPt*[G,E]): *FltPt*[G,E] Taylor series expansion for square root to third degree.

1.2 Acceleratable

This section lists operations which are primarily used for writing accelerator designs.

1.2.1 Annotations

Bound

The **bound** annotation provides a quick, simple way of annotating the approximate sizes of data consumed by the application. This is primarily useful in automated design space exploration, where the ideal parallelization factor or memory size may depend on the size of the data the accelerator will be processing.

The **bound** annotation is not a guarantee, so no operation optimizations are done as a result of this annotation. It is purely used to help drive runtime estimation.

While **bound** can be used outside of the *Accel* scopes, it currently has no use except in hardware tuning.

To set the bound of a given value, use the syntax:

```
bound(x) = 64
```

This tells the compiler to expect the value **x** to be approximately 64 at runtime.

DSE Parameters

To help drive automated design space exploration, Spatial includes syntax for users to explicitly annotate a value as a design parameter. These values are typically used to set the size of on-chip scratchpads like *SRAM* memories or to tune the parallelization/unrolling factors of *Controllers*.

Parameters have two parts: a default value that is used when compiler DSE is not run, and an associated range within which the compiler should explore.

There are two ways of creating explicit parameters in Spatial:

```
param(3)
```

This creates an integer parameter with a default value of 3. The range is left to be determined by the compiler. Alternatively, the user can set the range explicitly using:

```
1 (1 -> 10)
4 (1 -> 2 -> 10)
```

The first example shows a parameter with a default of 1, and a range of [1, 10]. The second example shows a parameter with a default of 4, and a range of {1, 2, 4, 8, 10} (stride of 2, including 1).

1.2.2 Controllers

Fold

Fold is a control node which takes an initial value and many scalar values and combines them into one value using some associative operator. Unless otherwise disabled, the compiler will attempt to automatically pipeline and parallelize *Fold* nodes. A Fold consists of a *map* function, which is responsible for producing the scalar values, and a *reduction* function to describe how the values should be combined.

Calling any form of Fold returns the accumulator which, at the end of the loop, will contain the final result of the reduction.

Static methods

object **Fold**
def **apply**[T](initial: T)(ctr: *Counter*)(map: *Int* => T)(reduce: (T,T) => T): Reg[T] Fold over a one dimensional space with implicit accumulator but explicit initial value. Returns the accumulator used to implement this reduction.
def **apply**[T](initial: T)(ctr1: *Counter*, ctr2: *Counter*)(map: (*Int*, *Int*) => T)(reduce: (T,T) => T): Reg[T] Fold over a two dimensional space with implicit accumulator but explicit initial value. Returns the accumulator used to implement this reduction.
def **apply**[T](initial: T)(ctr1: *Counter*, ctr2: *Counter*, ctr3: *Counter*)(map: (*Int*, *Int*, *Int*) => T)(reduce: (T,T) => T): Reg[T] Fold over a three dimensional space with implicit accumulator but explicit initial value. Returns the accumulator used to implement this reduction.
def **apply**[T](initial: T)(ctr1: *Counter*, ctr2: *Counter*, ctr3: *Counter*, ctr4: *Counter*, ctr5: *Counter**)(map: List[*Int*] => T)(reduce: (T,T) => T): Reg[T] Fold over an 4+ dimensional space with implicit accumulator but explicit initial value. Returns the accumulator used to implement this reduction. Note that the **map** function is on a **List** of iterators. The number of iterators will be the same as the number of counters supplied.
def **apply**[T](accum: Reg[T])(ctr: *Counter*)(map: *Int* => T)(reduce: (T,T) => T): Reg[T] Fold over a one dimensional space with explicit accumulator. Returns the accumulator used to implement this reduction.
def **apply**[T](accum: Reg[T])(ctr1: *Counter*, ctr2: *Counter*)(map: (*Int*, *Int*) => T)(reduce: (T,T) => T): Reg[T] Fold over a two dimensional space with explicit accumulator. Returns the accumulator used to implement this reduction.
def **apply**[T](accum: Reg[T])(ctr1: *Counter*, ctr2: *Counter*, ctr3: *Counter*)(map: (*Int*, *Int*, *Int*) => T)(reduce: (T,T) => T): Reg[T] Fold over a three dimensional space with explicit accumulator. Returns the accumulator used to implement this reduction.
def **apply**[T](accum: Reg[T])(ctr1: *Counter*, ctr2: *Counter*, ctr3: *Counter*, ctr4: *Counter*, ctr5: *Counter**)(map: List[*Int*] => T)(reduce: (T,T) => T): Reg[T] Fold over an 4+ dimensional space with explicit accumulator. Returns the accumulator used to implement this reduction. Note that the **map** function is on a **List** of iterators. The number of iterators will be the same as the number of counters supplied.

Foreach

The *Foreach* controller is similar to a *for* loop. Significantly, however, unless explicitly told otherwise, the compiler will assume each iteration of *Foreach* is independent, and will attempt to parallelize and pipeline the body. *Foreach* has no usable return value.

Static methods

object **Foreach**
def **apply**(ctr: *Counter*)(func: *Int* => *Unit*): *Unit* Foreach over a one dimensional space.
def **apply**(ctr1: *Counter*, ctr2: *Counter*)(func: (*Int*, *Int*) => *Unit*): *Unit* Foreach over a two dimensional space.
def **apply**(ctr1: *Counter*, ctr2: *Counter*, ctr3: *Counter*)(func: (*Int*, *Int*, *Int*) => *Unit*): *Unit* Foreach over a three dimensional space.
def **apply**(ctr1: *Counter*, ctr2: *Counter*, ctr3: *Counter*, ctr4: *Counter*, ctr5: *Counter**)(func: List[*Int*] => *Unit*): *Unit* Foreach over a 4+ dimensional space. Note that **func** is on a **List** of iterators. The number of iterators will be the same as the number of counters supplied.

MemFold

MemFold describes the reduction *across* multiple local memories. It functions essentially the same way as a *MemReduce*, except that it uses the existing value of the accumulator at the start of the reduction as the initial value of the accumulator, rather than resetting. Like MemReduce, unless otherwise disabled, the compiler will try to parallelize both the creation of multiple memories and the reduction of each of these memories into a single accumulator.

Static methods

object **MemFold**
def **apply**[T,C[T]](accum: C[T])(ctr: *Counter*)(map: *Int* => C[T])(reduce: (T,T) => T): C[T] On-chip memory fold over a one dimensional space. Returns the accumulator **accum**.
def **apply**[T,C[T]](accum: C[T])(ctr1: *Counter*, ctr2: *Counter*)(map: (*Int*, *Int*) => C[T])(reduce: (T,T) => T): C[T] On-chip memory fold over a two dimensional space. Returns the accumulator **accum**.
def **apply**[T,C[T]](accum: C[T])(ctr1: *Counter*, ctr2: *Counter*, ctr3: *Counter*)(map: (*Int*, *Int*, *Int*) => C[T])(reduce: (T,T) => T): C[T] On-chip memory fold over a three dimensional space. Returns the accumulator **accum**.
def **apply**[T,C[T]](accum: C[T])(ctr1: *Counter*, ctr2: *Counter*, ctr3: *Counter*, ctr4: *Counter*, ctr5: *Counter**)(map: List[*Int*] => C[T])(reduce: (T,T) => T): C[T] On-chip memory fold over a 4+ dimensional space. Returns the accumulator **accum**. Note that the **map** function is on a **List** of iterators. The number of iterators will be the same as the number of counters supplied.

MemReduce

MemReduce describes the reduction *across* multiple local memories. Like *Reduce*, MemReduce requires both a *map* and a *reduction* function. However, in MemReduce, the *map* describes the creation and population of a local memory (typically an *SRAM*). The *reduction* function still operates on scalars, and is used to combine local memories together element-wise. Unlike Reduce, MemReduce always requires an explicit accumulator. Unless otherwise disabled, the compiler will then try to parallelize both the creation of multiple memories and the reduction of each of these memories into a single accumulator.

Static methods

object **MemReduce**
def **apply**[T,C[T]](accum: C[T])(ctr: *Counter*)(map: *Int* => C[T])(reduce: (T,T) => T): C[T] On-chip memory reduction over a one dimensional space. Returns the accumulator **accum**.
def **apply**[T,C[T]](accum: C[T])(ctr1: *Counter*, ctr2: *Counter*)(map: (*Int*, *Int*) => C[T])(reduce: (T,T) => T): C[T] On-chip memory reduction over a two dimensional space. Returns the accumulator **accum**.
def **apply**[T,C[T]](accum: C[T])(ctr1: *Counter*, ctr2: *Counter*, ctr3: *Counter*)(map: (*Int*, *Int*, *Int*) => C[T])(reduce: (T,T) => T): C[T] On-chip memory reduction over a three dimensional space. Returns the accumulator **accum**.
def **apply**[T,C[T]](accum: C[T])(ctr1: *Counter*, ctr2: *Counter*, ctr3: *Counter*, ctr4: *Counter*, ctr5: *Counter**)(map: List[*Int*] => C[T])(reduce: (T,T) => T): C[T] On-chip memory reduction over a 4+ dimensional space. Returns the accumulator **accum**. Note that the **map** function is on a **List** of iterators. The number of iterators will be the same as the number of counters supplied.

Parallel

Parallel is a control node which informs the compiler to schedule any inner control nodes in a fork-join manner.

Note: Parallel will be soon be deprecated for general use as the scheduling algorithms in the Spatial compiler improve.

Static methods

object **Parallel**
def **apply**(body: => Unit): Unit Creates a parallel fork-join controller. Waits on completion of all controllers in the body.

Reduce

Reduce is a control node which takes many scalar values and combines them into one value using some associative operator. Unless otherwise disabled, the compiler will attempt to automatically pipeline and parallelize *Reduce* nodes. A Reduce consists of a *map* function, which is responsible for producing scalar values, and a *reduction* function to describe how the values should be combined.

Calling any form of Reduce returns the accumulator which, at the end of the loop, will contain the final result of the reduction.

Static methods

object **Reduce**
def **apply**[T](identity: T)(ctr: *Counter*)(map: *Int* => T)(reduce: (T,T) => T): Reg[T] Reduction over a one dimensional space with implicit accumulator but explicit identity value. Returns the accumulator used to implement this reduction.
def **apply**[T](identity: T)(ctr1: *Counter*, ctr2: *Counter*)(map: (*Int*, *Int*) => T)(reduce: (T,T) => T): Reg[T] Reduction over a two dimensional space with implicit accumulator but explicit identity value. Returns the accumulator used to implement this reduction.
def **apply**[T](identity: T)(ctr1: *Counter*, ctr2: *Counter*, ctr3: *Counter*)(map: (*Int*, *Int*, *Int*) => T)(reduce: (T,T) => T): Reg[T] Reduction over a three dimensional space with implicit accumulator but explicit identity value. Returns the accumulator used to implement this reduction.
def **apply**[T](identity: T)(ctr1: *Counter*, ctr2: *Counter*, ctr3: *Counter*, ctr4: *Counter*, ctr5: *Counter**)(map: List[*Int*] => T)(reduce: (T,T) => T): Reg[T] Reduction over an 4+ dimensional space with implicit accumulator but explicit identity value. Returns the accumulator used to implement this reduction. Note that the **map** function is on a **List** of iterators. The number of iterators will be the same as the number of counters supplied.
def **apply**[T](accum: Reg[T])(ctr: *Counter*)(map: *Int* => T)(reduce: (T,T) => T): Reg[T] Reduction over a one dimensional space with explicit accumulator. Returns the accumulator used to implement this reduction.
def **apply**[T](accum: Reg[T])(ctr1: *Counter*, ctr2: *Counter*)(map: (*Int*, *Int*) => T)(reduce: (T,T) => T): Reg[T] Reduction over a two dimensional space with explicit accumulator. Returns the accumulator used to implement this reduction.
def **apply**[T](accum: Reg[T])(ctr1: *Counter*, ctr2: *Counter*, ctr3: *Counter*)(map: (*Int*, *Int*, *Int*) => T)(reduce: (T,T) => T): Reg[T] Reduction over a three dimensional space with explicit accumulator. Returns the accumulator used to implement this reduction.
def **apply**[T](accum: Reg[T])(ctr1: *Counter*, ctr2: *Counter*, ctr3: *Counter*, ctr4: *Counter*, ctr5: *Counter**)(map: List[*Int*] => T)(reduce: (T,T) => T): Reg[T] Reduction over an 4+ dimensional space with explicit accumulator. Returns the accumulator used to implement this reduction. Note that the **map** function is on a **List** of iterators. The number of iterators will be the same as the number of counters supplied.

1.2.3 Off-Chip Memories

BufferedOut

BufferedOut defines an addressable output stream from the accelerator, implemented in hardware as an SRAM buffer which is asynchronously (and implicitly) used to drive a set of output pins.

BufferedOut is similar in functionality to *StreamOut*, but allows address and frame-based writing. This is useful, for example, in video processing, when video needs to be processed on a frame-by-frame basis rather than a pixel-by-pixel basis.

In Spatial, BufferedOuts are specified outside the Accel block, in host code.

Static methods

object **BufferedOut**
def **apply**[T:*Type*:*Bits*](bus: *Bus*): *BufferedOut*[T] Creates a BufferedOut of type T connected to the specified bus. The size of the buffer is currently fixed at 240 x 320 elements.

Infix methods

class **BufferedOut**[T]
def **update**(row: *Index*, col: *Index*, data: T): *Unit* Write **data** to the given two dimensional address.

Bus

In Spatial, **Buses** are predefined input/output pins available on supported FPGA targets. They are accessed using the static *target* object available in Spatial apps.

They currently have no corresponding syntax, and are used only to create *StreamIn* and *StreamOut* ports.

DRAM

DRAMs are pointers to locations in the accelerator's main memory comprising dense multi-dimensional arrays. They are the primary form of communication of data between the host and the accelerator. Data may be loaded to and from the accelerator in contiguous chunks (Tiles), or by bulk scatter and gather operations (SparseTiles).

Up to to 5-dimensional DRAMs are currently supported. Dimensionality of a DRAM instance is encoded by the subclass of DRAM. DRAM1, for instance represents a 1-dimensional DRAM.

A dense *DRAMDenseTile* can be created from a DRAM either using address range selection or by implicit conversion. When a Tile is created implicitly, it has the same address space as the entire original DRAM.

In Spatial, DRAMs are specified outside the Accel scope in the host code.

Static Methods

object **DRAM**

def **apply**[T:*Type*:*Bits*](length: *Index*): *DRAM1*[T]

Creates a reference to a 1-dimensional array in main memory with the given length.

Dimensions of a DRAM should be statically calculable functions of constants, parameters, and ArgIns.

def **apply**[T:*Type*:*Bits*](d1: *Index*, d2: *Index*): *DRAM2*[T]

Creates a reference to a 2-dimensional array in main memory with given rows and cols.

Dimensions of a DRAM should be statically calculable functions of constants, parameters, and ArgIns.

def **apply**[T:*Type*:*Bits*](d1: *Index*, d2: *Index*, d3: *Index*): *DRAM3*[T]

Creates a reference to a 3-dimensional array in main memory with given dimensions.

Dimensions of a DRAM should be statically calculable functions of constants, parameters, and ArgIns.

def **apply**[T:*Type*:*Bits*](d1: *Index*, d2: *Index*, d3: *Index*, d4: *Index*): *DRAM4*[T]

Creates a reference to a 4-dimensional array in main memory with given dimensions.

Dimensions of a DRAM should be statically calculable functions of constants, parameters, and ArgIns.

def **apply**[T:*Type*:*Bits*](d1: *Index*, d2: *Index*, d3: *Index*, d4: *Index*, d5: *Index*): *DRAM5*[T]

Creates a reference to a 5-dimensional array in main memory with given dimensions.

Dimensions of a DRAM should be statically calculable functions of constants, parameters, and ArgIns.

Infix methods

abstract class **DRAM**[T]

def **address**: *Int64*

Returns the 64-bit physical address in main memory of the start of this DRAM

def **dims**: List[*Index*]

Returns a Scala List of the dimensions of this DRAM

class **DRAM1**[T] extends DRAM[T]

def **size**: *Index*
Returns the total number of elements in this DRAM1.

def **length**: *Index*
Returns the total number of elements in this DRAM1.

def **apply**(range: *Range*): *DRAMDenseTile1*[T]
Creates a reference to a dense region of this DRAM1 for creating burst loads and stores.

def **apply**(addrs: *SRAM1*[*Index*]): *DRAMSparseTile*[T]
Creates a reference to a sparse region of this DRAM1 for use in scatter and gather transfers using all addresses in **addrs**.

def **apply**(addrs: *SRAM1*[*Index*], size: *Index*): *DRAMSparseTile*[T]
Creates a reference to a sparse region of this DRAM1 for use in scatter and gather transfers using the first **size** addresses in **addrs**.

def **apply**(addrs: *FIFO*[*Index*]): *DRAMSparseTile*[T]
Creates a reference to a sparse region of this DRAM1 for use in scatter and gather transfers using all addresses in **addrs**.

def **apply**(addrs: *FIFO*[*Index*], size: *Index*): *DRAMSparseTile*[T]
Creates a reference to a sparse region of this DRAM1 for use in scatter and gather transfers using the first **size** addresses in **addrs**.

def **apply**(addrs: *FILO*[*Index*]): *DRAMSparseTile*[T]
Creates a reference to a sparse region of this DRAM1 for use in scatter and gather transfers using all addresses in **addrs**.

def **apply**(addrs: *FILO*[*Index*], size: *Index*): *DRAMSparseTile*[T]
Creates a reference to a sparse region of this DRAM1 for use in scatter and gather transfers using the first **size** addresses in **addrs**.

def **store**(data: *SRAM1*[T]): *Unit*
Creates a dense, burst transfer from the given on-chip **data** to this DRAM's region of main memory.

def **store**(data: *FIFO*[T]): *Unit*
Creates a dense, burst transfer from the given on-chip **data** to this DRAM's region of main memory.

def **store**(data: *FILO*[T]): *Unit*
Creates a dense, burst transfer from the given on-chip **data** to this DRAM's region of main memory.

def **store**(data: *RegFile1*[T]): *Unit*
Creates a dense, burst transfer from the given on-chip **data** to this DRAM's region of main memory.

class **DRAM2**[T] extends DRAM[T]

def **rows**: *Index*

Returns the number of rows in this DRAM2

def **cols**: *Index*

Returns the number of columns in this DRAM2

def **size**: *Index*

Returns the total number of elements in this DRAM2

def **apply**(row: *Index*, cols: *Range*)

Creates a reference to a dense slice of a row of this DRAM2 for creating burst loads and stores.

def **apply**(rows: *Range*, col: *Index*)

Creates a reference to a dense slice of a column of this DRAM2 for creating burst loads and stores.

def **apply**(rows: *Range*, cols: *Range*)

Creates a reference to a 2-dimensional, dense region of this DRAM2 for creating burst loads and stores.

def **store**(sram: *SRAM2*[T]): *Unit*

Creates a dense, burst transfer from the given on-chip **data** to this DRAM's region of main memory.

def **store**(regs: *RegFile2*[T]): *Unit*

Creates a dense, burst transfer from the given on-chip **data** to this DRAM's region of main memory.

class **DRAM3**[T] extends DRAM[T]

def **dim0**: *Index*
Returns the first dimension for this DRAM3.

def **dim1**: *Index*
Returns the second dimension for this DRAM3.

def **dim2**: *Index*
Returns the third dimension for this DRAM3.

def **size**: *Index*
Returns the total number of elements in this DRAM3.

def **apply**(p: *Index*, r: *Index*, c: *Range*)
Creates a reference to a 1-dimensional, dense region of this DRAM3 for creating burst loads and stores.

def **apply**(p: *Index*, r: *Range*, c: *Index*)
Creates a reference to a 1-dimensional, dense region of this DRAM3 for creating burst loads and stores.

def **apply**(p: *Range*, r: *Index*, c: *Index*)
Creates a reference to a 1-dimensional, dense region of this DRAM3 for creating burst loads and stores.

def **apply**(p: *Index*, r: *Range*, c: *Range*)
Creates a reference to a 2-dimensional, dense region of this DRAM3 for creating burst loads and stores.

def **apply**(p: *Range*, r: *Index*, c: *Range*)
Creates a reference to a 2-dimensional, dense region of this DRAM3 for creating burst loads and stores.

def **apply**(p: *Range*, r: *Range*, c: *Index*)
Creates a reference to a 2-dimensional, dense region of this DRAM3 for creating burst loads and stores.

def **apply**(p: *Range*, r: *Range*, c: *Range*)
Creates a reference to a 3-dimensional, dense region of this DRAM3 for creating burst loads and stores.

def **store**(sram: *SRAM3*[T]): *Unit*
Creates a dense, burst transfer from the given on-chip **data** to this DRAM's region of main memory.

class **DRAM4**[T] extends DRAM[T]

def **dim0**: *Index*
Returns the first dimension of this DRAM4.

def **dim1**: *Index*
Returns the second dimension of this DRAM4.

def **dim2**: *Index*
Returns the third dimension of this DRAM4.

def **dim3**: *Index*
Returns the fourth dimension of this DRAM4.

def **size**: *Index*
Returns the total number of elements in this DRAM4.

def **apply**(q: *Index*, p: *Index*, r: *Index*, c: *Range*)
Creates a reference to a 1-dimensional, dense region of this DRAM4 for creating burst loads and stores.

def **apply**(q: *Index*, p: *Index*, r: *Range*, c: *Index*)
Creates a reference to a 1-dimensional, dense region of this DRAM4 for creating burst loads and stores.

def **apply**(q: *Index*, p: *Range*, r: *Index*, c: *Index*)
Creates a reference to a 1-dimensional, dense region of this DRAM4 for creating burst loads and stores.

def **apply**(q: *Range*, p: *Index*, r: *Index*, c: *Index*)
Creates a reference to a 1-dimensional, dense region of this DRAM4 for creating burst loads and stores.

def **apply**(q: *Index*, p: *Index*, r: *Range*, c: *Range*)
Creates a reference to a 2-dimensional, dense region of this DRAM4 for creating burst loads and stores.

def **apply**(q: *Range*, p: *Index*, r: *Index*, c: *Range*)
Creates a reference to a 2-dimensional, dense region of this DRAM4 for creating burst loads and stores.

def **apply**(q: *Range*, p: *Range*, r: *Index*, c: *Index*)
Creates a reference to a 2-dimensional, dense region of this DRAM4 for creating burst loads and stores.

def **apply**(q: *Index*, p: *Range*, r: *Index*, c: *Range*)
Creates a reference to a 2-dimensional, dense region of this DRAM4 for creating burst loads and stores.

def **apply**(q: *Range*, p: *Index*, r: *Range*, c: *Index*)
Creates a reference to a 2-dimensional, dense region of this DRAM4 for creating burst loads and stores.

def **apply**(q: *Index*, p: *Range*, r: *Range*, c: *Index*)
Creates a reference to a 2-dimensional, dense region of this DRAM4 for creating burst loads and stores.

def **apply**(q: *Index*, p: *Range*, r: *Range*, c: *Range*)
Creates a reference to a 3-dimensional, dense region of this DRAM4 for creating burst loads and stores.

def **apply**(q: *Range*, p: *Index*, r: *Range*, c: *Range*)
Creates a reference to a 3-dimensional, dense region of this DRAM4 for creating burst loads and stores.

def **apply**(q: *Range*, p: *Range*, r: *Index*, c: *Range*)
Creates a reference to a 3-dimensional, dense region of this DRAM4 for creating burst loads and stores.

def **apply**(q: *Range*, p: *Range*, r: *Range*, c: *Index*)

class **DRAM5**[T] extends DRAM[T]

def **dim0**: *Index*
Returns the first dimension of this DRAM5.

def **dim1**: *Index*
Returns the second dimension of this DRAM5.

def **dim2**: *Index*
Returns the third dimension of this DRAM5.

def **dim3**: *Index*
Returns the fourth dimension of this DRAM5.

def **dim4**: *Index*
Returns the fifth dimension of this DRAM5.

def **size**: *Index*
Returns the total number of elements in this DRAM5.

def **apply**(x: *Index*, q: *Index*, p: *Index*, r: *Index*, c: *Range*): *DRAMDenseTile1*[T]
Creates a reference to a 1-dimensional, dense region of this DRAM5 for creating burst loads and stores.

def **apply**(x: *Index*, q: *Index*, p: *Index*, r: *Range*, c: *Index*): *DRAMDenseTile1*[T]
Creates a reference to a 1-dimensional, dense region of this DRAM5 for creating burst loads and stores.

def **apply**(x: *Index*, q: *Index*, p: *Range*, r: *Index*, c: *Index*): *DRAMDenseTile1*[T]
Creates a reference to a 1-dimensional, dense region of this DRAM5 for creating burst loads and stores.

def **apply**(x: *Index*, q: *Range*, p: *Index*, r: *Index*, c: *Index*): *DRAMDenseTile1*[T]
Creates a reference to a 1-dimensional, dense region of this DRAM5 for creating burst loads and stores.

def **apply**(x: *Range*, q: *Index*, p: *Index*, r: *Index*, c: *Index*): *DRAMDenseTile1*[T]
Creates a reference to a 1-dimensional, dense region of this DRAM5 for creating burst loads and stores.

def **apply**(x: *Index*, q: *Index*, p: *Index*, r: *Range*, c: *Range*): *DRAMDenseTile2*[T]
Creates a reference to a 2-dimensional, dense region of this DRAM5 for creating burst loads and stores.

def **apply**(x: *Index*, q: *Index*, p: *Range*, r: *Index*, c: *Range*): *DRAMDenseTile2*[T]
Creates a reference to a 2-dimensional, dense region of this DRAM5 for creating burst loads and stores.

def **apply**(x: *Index*, q: *Index*, p: *Range*, r: *Range*, c: *Index*): *DRAMDenseTile2*[T]
Creates a reference to a 2-dimensional, dense region of this DRAM5 for creating burst loads and stores.

def **apply**(x: *Index*, q: *Range*, p: *Index*, r: *Index*, c: *Range*): *DRAMDenseTile2*[T]
Creates a reference to a 2-dimensional, dense region of this DRAM5 for creating burst loads and stores.

def **apply**(x: *Index*, q: *Range*, p: *Index*, r: *Range*, c: *Index*): *DRAMDenseTile2*[T]
Creates a reference to a 2-dimensional, dense region of this DRAM5 for creating burst loads and stores.

Continued on next page

Table 1.2 – continued from previous page

class **DRAM5**[T] extends DRAM[T]
def **apply**(x: *Index*, q: *Range*, p: *Range*, r: *Index*, c: *Index*): *DRAMDenseTile2*[T] Creates a reference to a 2-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Range*, q: *Index*, p: *Index*, r: *Index*, c: *Range*): *DRAMDenseTile2*[T] Creates a reference to a 2-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Range*, q: *Index*, p: *Index*, r: *Range*, c: *Index*): *DRAMDenseTile2*[T] Creates a reference to a 2-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Range*, q: *Index*, p: *Range*, r: *Index*, c: *Index*): *DRAMDenseTile2*[T] Creates a reference to a 2-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Range*, q: *Range*, p: *Index*, r: *Index*, c: *Index*): *DRAMDenseTile2*[T] Creates a reference to a 2-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Index*, q: *Index*, p: *Range*, r: *Range*, c: *Range*): *DRAMDenseTile3*[T] Creates a reference to a 3-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Index*, q: *Range*, p: *Index*, r: *Range*, c: *Range*): *DRAMDenseTile3*[T] Creates a reference to a 3-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Index*, q: *Range*, p: *Range*, r: *Index*, c: *Range*): *DRAMDenseTile3*[T] Creates a reference to a 3-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Index*, q: *Range*, p: *Range*, r: *Range*, c: *Index*): *DRAMDenseTile3*[T] Creates a reference to a 3-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Range*, q: *Index*, p: *Index*, r: *Range*, c: *Range*): *DRAMDenseTile3*[T] Creates a reference to a 3-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Range*, q: *Index*, p: *Range*, r: *Index*, c: *Range*): *DRAMDenseTile3*[T] Creates a reference to a 3-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Range*, q: *Index*, p: *Range*, r: *Range*, c: *Index*): *DRAMDenseTile3*[T] Creates a reference to a 3-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Range*, q: *Range*, p: *Index*, r: *Index*, c: *Range*): *DRAMDenseTile3*[T] Creates a reference to a 3-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Range*, q: *Range*, p: *Index*, r: *Range*, c: *Index*): *DRAMDenseTile3*[T] Creates a reference to a 3-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Range*, q: *Range*, p: *Range*, r: *Index*, c: *Index*): *DRAMDenseTile3*[T] Creates a reference to a 3-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Index*, q: *Range*, p: *Range*, r: *Range*, c: *Range*): *DRAMDenseTile4*[T] Creates a reference to a 4-dimensional, dense region of this DRAM5 for creating burst loads and stores.

Continued on next page

Table 1.2 – continued from previous page

class **DRAM5**[T] extends DRAM[T]
def **apply**(x: *Range*, q: *Index*, p: *Range*, r: *Range*, c: *Range*): *DRAMDenseTile4*[T] Creates a reference to a 4-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Range*, q: *Range*, p: *Index*, r: *Range*, c: *Range*): *DRAMDenseTile4*[T] Creates a reference to a 4-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Range*, q: *Range*, p: *Range*, r: *Index*, c: *Range*): *DRAMDenseTile4*[T] Creates a reference to a 4-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Range*, q: *Range*, p: *Range*, r: *Range*, c: *Index*): *DRAMDenseTile4*[T] Creates a reference to a 4-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **apply**(x: *Range*, q: *Range*, p: *Range*, r: *Range*, c: *Range*): *DRAMDenseTile5*[T] Creates a reference to a 5-dimensional, dense region of this DRAM5 for creating burst loads and stores.
def **store**(data: *SRAM5*[T]): *Unit* Creates a dense, burst transfer from the given on-chip **data** to this DRAM's region of main memory.

Implicit methods

def **createTile1**[T](dram: *DRAM1*[T]): *DRAMDenseTile1*[T] Implicitly converts a DRAM1 to a DRAMDenseTile1 with the same address space.
def **createTile2**[T](dram: *DRAM2*[T]): *DRAMDenseTile2*[T] Implicitly converts a DRAM2 to a DRAMDenseTile2 with the same address space.
def **createTile3**[T](dram: *DRAM3*[T]): *DRAMDenseTile3*[T] Implicitly converts a DRAM3 to a DRAMDenseTile3 with the same address space.
def **createTile4**[T](dram: *DRAM4*[T]): *DRAMDenseTile4*[T] Implicitly converts a DRAM4 to a DRAMDenseTile4 with the same address space.
def **createTile2**[T](dram: *DRAM5*[T]): *DRAMDenseTile5*[T] Implicitly converts a DRAM5 to a DRAMDenseTile5 with the same address space.

DRAMDenseTile

A **DRAMDenseTile** describes a continguous slice of a *DRAM* memory's address space which can be loaded onto the accelerator for processing or which can be updated with results once FPGA computation is complete.

Infix methods

class **DRAMDenseTile**

class **DRAMDenseTile1**[T] extends DRAMDenseTile[T]

def **store**(data: *SRAM1*[T]): *Unit*

Creates a dense, burst transfer from the given on-chip **data** to this tiles's region of main memory.

def **store**(data: *FIFO*[T]): *Unit*

Creates a dense, burst transfer from the given on-chip **data** to this tiles's region of main memory.

def **store**(data: *FILO*[T]): *Unit*

Creates a dense, burst transfer from the given on-chip **data** to this tiles's region of main memory.

def **store**(data: *RegFile1*[T]): *Unit*

Creates a dense, burst transfer from the given on-chip **data** to this tiles's region of main memory.

class **DRAMDenseTile2**[T] extends DRAMDenseTile[T]

def **store**(data: *SRAM2*[T]): *Unit*

Creates a dense, burst transfer from the given on-chip **data** to this tiles's region of main memory.

def **store**(data: *RegFile2*[T]): *Unit*

Creates a dense, burst transfer from the given on-chip **data** to this tiles's region of main memory.

class **DRAMDenseTile3**[T] extends DRAMDenseTile[T]

def **store**(data: *SRAM3*[T]): *Unit*

Creates a dense, burst transfer from the given on-chip **data** to this tiles's region of main memory.

class **DRAMDenseTile4**[T] extends DRAMDenseTile[T]

def **store**(data: *SRAM4*[T]): *Unit*

Creates a dense, burst transfer from the given on-chip **data** to this tiles's region of main memory.

class **DRAMDenseTile5**[T] extends DRAMDenseTile[T]

def **store**(data: *SRAM5*[T]): *Unit*

Creates a dense, burst transfer from the given on-chip **data** to this tiles's region of main memory.

DRAMSparseTile

A **DRAMSparseTile** describes a sparse section of a DRAM memory which can be loaded onto the accelerator using a gather operation, or which can be updated using a scatter operation.

Infix methods

class **DRAMSparseTile**
def **scatter**(data: *SRAM1*[T]): *Unit* Creates a sparse transfer from the given on-chip **data** to this sparse region of main memory.
def **scatter**(data: *FIFO*[T]): *Unit* Creates a sparse transfer from the given on-chip **data** to this sparse region of main memory.
def **scatter**(data: *FILO*[T]): *Unit* Creates a sparse transfer from the given on-chip **data** to this sparse region of main memory.

StreamIn

StreamIn defines a hardware bus used to receive streaming data from outside of the FPGA. StreamIns may not be written to. For streaming outputs, use *StreamOut*. StreamIns are specified using a *Bits* - based type and a target *Bus*.

In Spatial, StreamIns are specified outside the Accel block, in host code.

Static methods

object **StreamIn**
def **apply**[T:*Type*:*Bits*](bus: *Bus*): *StreamIn*[T] Creates a StreamIn of type T connected to the specified **bus** pins.

Infix methods

class **StreamIn**[T]
def **value**: T Returns the current value of this StreamIn. Equivalent in hardware to connecting a wire to the StreamIn's bus.

Implicit methods

def **readStream**[T](stream: *StreamIn*[T]): T Implicitly creates a wire connecting to the StreamIn's bus.

StreamOut

StreamOut defines a hardware bus used to output streaming data from the FPGA. StreamOuts may not be read from. For streaming inputs, use *StreamIn*. StreamOuts are specified using a *Bits* - based type and a target *Bus*.

In Spatial, StreamOuts are specified outside the Accel block, in host code.

Static methods

object **StreamOut**
def **apply**[T:*Type*:*Bits*](bus: *Bus*): *StreamOut*[T] Creates a StreamOut of type T connected to the specified target bus pins.

Infix methods

class **StreamOut**[T]
def **:=**(data: T): *Unit* Connect the given **data** to this StreamOut.
def **:=**(data: T, en: *Bit*): *Unit* Connect the given **data** to this StreamOut with enable **en**.

1.2.4 On-Chip Memories

Spatial includes a variety of on-chip memory templates for use in accelerator design. Unless otherwise specified, the size of a specified on-chip memory allocation must always be a statically calculable number in order to map it to hardware resources.

Additionally, on-chip allocations follow traditional software scoping rules. Allocations done within the scope of *Controllers*, for example, logically create new memories on each loop iteration. Unless otherwise specified, the contents of on-chip memories are undefined upon allocation in order to minimize unnecessary memory initialization.

Counter

Counter is a single hardware counter with an associated start (inclusive), end (exclusive), step size, and parallelization factor. By default, the parallelization factor is assumed to be a design parameter. Counters can be chained together using CounterChain, but this is typically done implicitly when creating controllers.

It is generally recommended to create a *Range* and allow the compiler to implicitly convert this to a Counter, as Range provides slightly better syntax sugar.

Static Methods

object **Counter**
def **apply**(end: *Index*): *Counter* Creates a Counter with start of 0, given **end**, and step size of 1.
def **apply**(start: *Index*, end: *Index*): *Counter* Creates a Counter with given **start** and **end**, and step size of 1.
def **apply**(start: *Index*, end: *Index*, step: *Index*): *Counter* Creates a Counter with given **start**, **end**, and **step** size.
def **apply**(start: *Index*, end: *Index*, step: *Index*, par: *Index*): *Counter* Creates a Counter with given **start**, **end**, **step**, and **par** parallelization factor.

CounterChain

CounterChain describes a set of chained hardware counters, where a given counter increments only when the counter below it wraps around. Order is specified as outermost counter first to innermost counter last.

CounterChains are generally created implicitly for small numbers of counters, and need only be created explicitly for more than 3 chained counters.

Static methods

object **CounterChain**
def **apply**(counters: *Counter**): *CounterChain* Creates a chain of counters. Order is specified as outermost on the left to innermost on the right.

FIFO

FIFOs (first-in, first-out) are on-chip scratchpads with additional control logic for address-less enqueue/dequeue operations. FIFOs preserve the ordering between elements as they are enqueued. A FIFO's **deq** operation always returns the oldest **enqueued** element which has not yet been dequeued.

Static methods

object **FIFO**
def **apply**[T:*Type*:*Bits*](depth: *Int*): *FIFO*[T] Creates a FIFO with given **depth**. Depth must be a statically determinable signed integer.

Infix methods

class **FIFO**[T]
def **par**(p: *Index*): *FIFO*[T] = { val x = *FIFO*(s); x.p Annotates that addresses in this FIFO can be read in parallel by factor **p**. Used when creating references to sparse regions of DRAM.
def **empty**(): *Bit* Returns true when this FIFO contains no elements, false otherwise.
def **full**(): *Bit* Returns true when this FIFO cannot fit any more elements, false otherwise.
def **almostEmpty**(): *Bit* Returns true when this FIFO contains exactly one element, false otherwise.
def **almostFull**(): *Bit* Returns true when this FIFO can fit exactly one more element, false otherwise.
def **numel**(): *Index* Returns the number of elements currently in this FIFO.
def **enq**(data: T): *Unit* Creates an enqueue (write) port of **data** to this FIFO.
def **enq**(data: T, en: *Bit*): *Unit* Creates an enqueue (write) port of **data** to this FIFO, enabled by **en**.
def **deq**(): T Creates a dequeue (destructive read) port from this FIFO.
def **deq**(en: *Bit*): T Creates a dequeue (destructive read) port from this FIFO, enabled by **en**.
def **peek**(): T Creates a non-destructive read port from this FIFO.
def **load**(dram: *DRAMDenseTile1*[T]): *Unit* Creates a dense, burst load from the specified region of DRAM to this on-chip memory.
def **gather**(dram: *DRAMSparseTile*[T]): *Unit* Creates a sparse load from the specified sparse region of DRAM to this on-chip memory.

FILO

FILOs (first-in, last-out) are on-chip scratchpads with additional control logic for address-less enqueue/dequeue operations. FILOs acts as a Stack, reversing the order of elements it receives. A FILO's **pop** operation always returns the most recently **pushed** element.

Static methods

object **FILO**

def **apply**[T:*Type*:*Bits*](size: *Index*): *FILO*[T]

Creates a FILO with given **depth**.

Depth must be a statically determinable signed integer.

Infix methods

class **FILO**[T]
def **par**(p: *Index*): *FILO*[T] = { val x = *FILO*(s); x.p Annotates that addresses in this FIFO can be read in parallel by factor **p**. Used when creating references to sparse regions of DRAM.
def **empty**(): *Bit* Returns true when this FILO contains no elements, false otherwise.
def **full**(): *Bit* Returns true when this FILO cannot fit any more elements, false otherwise.
def **almostEmpty**(): *Bit* Returns true when this FILO contains exactly one element, false otherwise.
def **almostFull**(): *Bit* Returns true when this FILO can fit exactly one more element, false otherwise.
def **numel**(): *Index* Returns the number of elements currently in this FILO.
def **push**(data: T): *Unit* Creates a push (write) port to this FILO of **data**.
def **push**(data: T, en: *Bit*): *Unit* Creates a conditional push (write) port to this FILO of **data** enabled by **en**.
def **pop**(): T Creates a pop (destructive read) port to this FILO.
def **pop**(en: *Bit*): T Creates a conditional pop (destructive read) port to this FILO enabled by **en**.
def **peek**(): T Creates a non-destructive read port to this FILO.
def **load**(dram: *DRAMDenseTile1*[T]): *Unit* Creates a dense, burst load from the specified region of DRAM to this on-chip memory.
def **gather**(dram: *DRAMSparseTile*[T]): *Unit* Creates a sparse load from the specified sparse region of DRAM to this on-chip memory.

LineBuffer

LineBuffers are two dimensional, on-chip scratchpads with a fixed size. LineBuffers act as a FIFO on input, supporting only queued writes, but support addressed reading like SRAMs. For writes, the current row buffer and column is maintained using an internal counter. This counter resets every time the controller containing the enqueue completes execution.

The contents of LineBuffers are persistent across loop iterations, even when they are declared in an inner scope. Up to 5-dimensional LineBuffers are currently supported.

Static methods

object **LineBuffer**
def **apply**[T:*Type*:*Bits*](rows: *Index*, cols: *Index*): *LineBuffer*[T] Allocates a LineBuffer with given **rows** and **cols**. The contents of this LineBuffer are initially undefined. **rows** and **cols** must be statically determinable integers.
def **strided**[T:*Type*:*Bits*](rows: *Index*, cols: *Index*, stride: *Index*): *LineBuffer*[T] Allocates a LineBuffer with given number of **rows** and **cols**, and with given **stride**. The contents of this LineBuffer are initially undefined. **rows**, **cols**, and **stride** must be statically determinable integers.

Infix methods

class **LineBuffer**[T]
def **apply**(row: *Index*, col: *Index*): T Creates a load port to this LineBuffer at the given **row** and **col**.
def **apply**(row: *Index*, cols: *Range*)(implicit ctx: SrcCtx): *Vector*[T] Creates a vectorized load port to this LineBuffer at the given **row** and **cols**.
def **apply**(rows: *Range*, col: *Index*)(implicit ctx: SrcCtx): *Vector*[T] Creates a vectorized load port to this LineBuffer at the given **rows** and **col**.
def **enq**(data: T): *Unit* Creates an enqueue (write) port of **data** to this LineBuffer.
def **enq**(data: T, en: *Bit*): *Unit* Creates an enqueue (write) port of **data** to this LineBuffer, enabled by **en**.
def **load**(dram: *DRAMDenseTile1*[T])(implicit ctx: SrcCtx): *Unit* Creates a dense transfer from the given region of DRAM to this on-chip memory.

LUT

LUTs are on-chip, read-only memories of fixed size. LUTs can be specified as 1 to 5 dimensional.

LUTs can be created from files using the `fromFile` methods in the LUT object. This file reading is currently done at compilation time.

Static methods

object **LUT**
def **apply**[T:*Type*:*Bits*](dim: *Int*)(elems: T*): *LUT1*[T] Creates a 1-dimensional read-only lookup table with given elements. The number of supplied elements must match the given dimensions.
def **apply**[T:*Type*:*Bits*](dim1: *Int*, dim2: *Int*)(elems: T*): *LUT2*[T] Creates a 2-dimensional read-only lookup table with given elements. The number of supplied elements must match the given dimensions.
def **apply**[T:*Type*:*Bits*](dim1: *Int*, dim2: *Int*, dim3: *Int*)(elems: T*): *LUT3*[T] Creates a 3-dimensional read-only lookup table with given elements. The number of supplied elements must match the given dimensions.
def **apply**[T:*Type*:*Bits*](dim1: *Int*, dim2: *Int*, dim3: *Int*, dim4: *Int*)(elems: T*): *LUT4*[T] Creates a 4-dimensional read-only lookup table with given elements. The number of supplied elements must match the given dimensions.
def **apply**[T:*Type*:*Bits*](dim1: *Int*, dim2: *Int*, dim3: *Int*, dim4: *Int*, dim5: *Int*)(elems: T*): *LUT5*[T] Creates a 5-dimensional read-only lookup table with given elements. The number of supplied elements must match the given dimensions.
def **fromFile**[T:*Type*:*Bits*](dim1: *Int*)(filename: *String*): *LUT1*[T] Creates a 1-dimensional read-only lookup table from the given data file. Note that this file is read during **compilation**, not runtime. The number of supplied elements must match the given dimensions.
def **fromFile**[T:*Type*:*Bits*](dim1: *Int*, dim2: *Int*)(filename: *String*): *LUT2*[T] Creates a 2-dimensional read-only lookup table from the given data file. Note that this file is read during **compilation**, not runtime. The number of supplied elements must match the given dimensions.
def **fromFile**[T:*Type*:*Bits*](dim1: *Int*, dim2: *Int*, dim3: *Int*)(filename: *String*): *LUT3*[T] Creates a 3-dimensional read-only lookup table from the given data file. Note that this file is read during **compilation**, not runtime. The number of supplied elements must match the given dimensions.
def **fromFile**[T:*Type*:*Bits*](dim1: *Int*, dim2: *Int*, dim3: *Int*, dim4: *Int*)(filename: *String*): *LUT4*[T] Creates a 4-dimensional read-only lookup table from the given data file. Note that this file is read during **compilation**, not runtime. The number of supplied elements must match the given dimensions.
def **fromFile**[T:*Type*:*Bits*](dim1: *Int*, dim2: *Int*, dim3: *Int*, dim4: *Int*, dim5: *Int*)(filename: *String*): *LUT5*[T] Creates a 5-dimensional read-only lookup table from the given data file. Note that this file is read during **compilation**, not runtime. The number of supplied elements must match the given dimensions.

Infix methods

abstract class **LUT**[T]

class **LUT1**[T] extends LUT[T]

def **apply**(i: *Index*): T

Returns the element at the given address **i**.

class **LUT2**[T] extends LUT[T]

def **apply**(r: *Index*, c: *Index*): T

Returns the element at the given address **r**, **c**.

class **LUT3**[T] extends LUT[T]

def **apply**(r: *Index*, c: *Index*, p: *Index*): T

Returns the element at the given 3-dimensional address.

class **LUT4**[T] extends LUT[T]

def **apply**(r: *Index*, c: *Index*, p: *Index*, q: *Index*): T

Returns the element at the given 4-dimensional address.

class **LUT5**[T] extends LUT[T]

def **apply**(r: *Index*, c: *Index*, p: *Index*, q: *Index*, m: *Index*): T

Returns the element at the given 5-dimensional address.

Reg

Reg defines a hardware register used to hold a scalar value. The default reset value for a Reg is the numeric zero value for it's specified type.

ArgIn, **ArgOut**, and **HostIO** are specialized forms of Reg which are used to transfer scalar values to and from the accelerator. ArgIns and ArgOuts are used for setup values at the initialization of the FPGA. ArgIns may not be written to, while ArgOuts generally should not be read from. HostIOs are for values which may be continuously changed or read by the host during FPGA execution.

In Spatial, ArgIns, ArgOuts, and HostIO registers are specified outside the Accel block, in host code.

Static methods

object **Reg**

def **apply**[T:*Type*:*Bits*]: Reg[T]

Creates a register of type T with a reset value of zero.

def **apply**[T:*Type*:*Bits*](reset: T): Reg[T]

Creates a register of type T with the given **reset** value.

object **ArgIn**

def **apply**[T:*Type*:*Bits*]: Reg[T]

Creates an input argument register of type T with a reset value of zero.

object **ArgOut**

def **apply**[T:*Type*:*Bits*]: Reg[T]

Creates an output argument register of type T with a reset value of zero.

object **HostIO**

def **apply**[T:*Type*:*Bits*]: Reg[T]

Creates a host I/O register of type T with a reset value of zero.

Infix methods

class **Reg**[T]

def **value**: T

Returns the value currently held by this register.

def **:=**(data: T): *Unit*

Writes the given **data** to this register.

def **reset**: *Unit*

Resets the value of this register back to its reset value.

def **reset**(cond: *Bit*): *Unit*

Conditionally resets the value of this register back to its reset value if **cond** is true.

Implicit methods

def **readReg**[T](reg: Reg[T]): T

Implicitly reads the value of this register.

RegFile

RegFiles are on-chip arrays of registers with fixed size. RegFiles currently can be specified as one or two dimensional. Like other memories in Spatial, the contents of RegFiles are persistent across loop iterations, even when they are declared in an inner scope.

Using the **<<=** operator, RegFiles can be used as shift registers. 2-dimensional RegFiles must select a specific row or column before shifting using *regfile(row, *)* or *regfile(*, col)*, respectively.

Static methods

object **RegFile**
def **apply**[T:*Type*:*Bits*](length: *Index*): *RegFile1*[T] Allocates a 1-dimensional Regfile with specified **length**.
def **apply**[T:*Type*:*Bits*](length: *Int*, inits: List[T]): *RegFile1*[T] Allocates a 1-dimensional RegFile with specified **length** and initial values **inits**. The number of initial values must be the same as the total size of the RegFile.
def **apply**[T:*Type*:*Bits*](rows: *Index*, cols: *Index*): *RegFile2*[T] Allocates a 2-dimensional RegFile with specified **rows** and **cols**.
def **apply**[T:*Type*:*Bits*](rows: *Int*, cols: *Int*, inits: List[T]): *RegFile2*[T] Allocates a 2-dimensional RegFile with specified **rows** and **cols** and initial values **inits**. The number of initial values must be the same as the total size of the RegFile
def **apply**[T:*Type*:*Bits*](dim0: *Index*, dim1: *Index*, dim2: *Index*): *RegFile3*[T] Allocates a 3-dimensional RegFile with specified dimensions.
def **apply**[T:*Type*:*Bits*](dim0: *Int*, dim1: *Int*, dim2: *Int*, inits: List[T]): *RegFile3*[T] Allocates a 3-dimensional RegFile with specified dimensions and initial values **inits**. The number of initial values must be the same as the total size of the RegFile

---

Infix methods

abstract class **RegFile**[T]
def **reset**: *Unit* Resets this RegFile to its initial values (or zeros, if unspecified).
def **reset**(cond: *Bit*): *Unit* Conditionally resets this RegFile based on **cond** to its inital values (or zeros if unspecified).

class **RegFile1**[T] extends RegFile[T]

def **apply**(i: *Index*): T

Returns the value held by the register at address **i**.

def **update**(i: *Index*, data: T): *Unit*

Updates the register at address **i** to hold **data**.

def **<<=**(data: T): *Unit*

Shifts in **data** into the first register, shifting all other values over by one position.

def **<<=**(data: *Vector*[T]): *Unit*

Shifts in **data** into the first N registers, where N is the size of the given Vector.

All other elements are shifted by N positions.

def **load**(dram: *DRAM1*[T]): *Unit*

Creates a dense, burst load from the specified region of DRAM to this on-chip memory.

def **load**(dram: *DRAMDenseTile1*[T]): *Unit*

Creates a dense, burst load from the specified region of DRAM to this on-chip memory.

class **RegFile2**[T] extends RegFile[T]

def **apply**(r: *Index*, c: *Index*): T

Returns the value held by the register at row **r**, column **c**.

def **update**(r: *Index*, c: *Index*, data: T): *Unit*

Updates the register at row **r**, column **c** to hold the given **data**.

def **apply**(i: *Index*, y: Wildcard)

Returns a view of row **i** of this RegFile.

def **apply**(y: Wildcard, i: *Index*)

Returns a view of column **i** of this RegFile.

def **load**(dram: *DRAM2*[T]): *Unit*

Creates a dense, burst load from the specified region of DRAM to this on-chip memory.

def **load**(dram: *DRAMDenseTile2*[T]): *Unit*

Creates a dense, burst load from the specified region of DRAM to this on-chip memory.

class **RegFile3**[T] extends RegFile[T]
def **apply**(dim0: *Index*, dim1: *Index*, dim2: *Index*): T Returns the value held by the register at the given 3-dimensional address.
def **update**(dim0: *Index*, dim1: *Index*, dim2: *Index*, data: T): *Unit* Updates the register at the given 3-dimensional address to hold the given **data**.
def **apply**(i: *Index*, j: *Index*, y: Wildcard) Returns a 1-dimensional view of part of this RegFile3.
def **apply**(i: *Index*, y: Wildcard, j: *Index*) Returns a 1-dimensional view of part of this RegFile3.
def **apply**(y: Wildcard, i: *Index*, j: *Index*) Returns a 1-dimensional view of part of this RegFile3.
def **load**(dram: *DRAM3*[T]): *Unit* Creates a dense, burst load from the specified region of DRAM to this on-chip memory.
def **load**(dram: *DRAMDenseTile3*[T]): *Unit* Creates a dense, burst load from the specified region of DRAM to this on-chip memory.

SRAM

SRAMs are on-chip scratchpads with fixed size. SRAMs can be specified as multi-dimensional, but the underlying addressing in hardware is always flat. The contents of SRAMs are persistent across loop iterations, even when they are declared in an inner scope. Up to 5-dimensional SRAMs are currently supported.

Static methods

object **SRAM**
def **apply**[T:*Type*:*Bits*](length: *Index*): *SRAM1*[T] Allocates a 1-dimensional SRAM with the specified **length**.
def **apply**[T:*Type*:*Bits*](rows: *Index*, cols: *Index*): *SRAM2*[T] Allocates a 2-dimensional SRAM with the specified number of **rows** and **cols**.
def **apply**[T:*Type*:*Bits*](p: *Index*, r: *Index*, c: *Index*): *SRAM3*[T] Allocates a 3-dimensional SRAM with the specified dimensions.
def **apply**[T:*Type*:*Bits*](q: *Index*, p: *Index*, r: *Index*, c: *Index*): *SRAM4*[T] Allocates a 4-dimensional SRAM with the specified dimensions.
def **apply**[T:*Type*:*Bits*](m: *Index*, q: *Index*, p: *Index*, r: *Index*, c: *Index*): *SRAM5*[T] Allocates a 5-dimensional SRAM with the specified dimensions.

Infix methods

abstract class **SRAM**[T]

def **dims**: List[*Index*]
Returns a Scala List of the dimensions of this DRAM

class **SRAM1**[T] extends SRAM[T]

def **length**: *Index*
Returns the total size of this SRAM1.

def **size**: *Index*
Returns the total size of this SRAM1.

def **par**(p: *Index*): *SRAM1*[T] = { val x = *SRAM1*(s); x.p
Annotates that addresses in this SRAM1 can be read in parallel by factor **p**.

Used when creating references to sparse regions of DRAM.

def **apply**(a: *Index*): T
Returns the value in this SRAM1 at the given address **a**.

def **update**(a: *Index*, data: T): *Unit*
Updates the value in this SRAM1 at the given address **a** to **data**.

def **gather**(dram: *DRAMSparseTile*[T]): *Unit*
Create a sparse load from the given sparse region of DRAM to this on-chip memory.

Elements will be gathered and stored contiguously in this memory.

def **load**(dram: *DRAM1*[T]): *Unit*
Create a dense, burst load from the given region of DRAM to this on-chip memory.

def **load**(dram: *DRAMDenseTile1*[T]): *Unit*
Create a dense, burst load from the given region of DRAM to this on-chip memory.

class **SRAM2**[T] extends SRAM[T]

def **rows**: *Index*

Returns the number of rows in this SRAM2.

def **cols**: *Index*

Returns the number of columns in this SRAM2.

def **size**: *Index*

Returns the total size of this SRAM2.

def **apply**(row: *Index*, col: *Index*): T

Returns the value in this SRAM2 at the given **row** and **col**.

def **update**(row: *Index*, col: *Index*, data: T): *Unit*

Updates the value in this SRAM2 at the given **row** and **col** to **data**.

def **par**(p: *Index*): *SRAM2*[T] = { val x = *SRAM2*(s); x.p

Annotates that addresses in this SRAM2 can be read in parallel by factor **p**.

Used when creating references to sparse regions of DRAM.

def **load**(dram: *DRAM2*[T]): *Unit*

Create a dense, burst load from the given region of DRAM to this on-chip memory.

def **load**(dram: *DRAMDenseTile2*[T]): *Unit*

Create a dense, burst load from the given region of DRAM to this on-chip memory.

class **SRAM3**[T] extends SRAM[T]

def **dim0**: *Index*
Returns the first dimension of this SRAM3.

def **dim1**: *Index*
Returns the second dimension of this SRAM3.

def **dim2**: *Index*
Returns the third dimension of this SRAM3.

def **size**: *Index*
Returns the total size of this SRAM3.

def **apply**(a: *Index*, b: *Index*, c: *Index*): T
Returns the value in this SRAM3 at the given 3-dimensional address **a**, **b**, **c**.

def **update**(a: *Index*, b: *Index*, c: *Index*, data: T): *Unit*
Updates the value in this SRAM3 at the given 3-dimensional address to **data**.

def **par**(p: *Index*): *SRAM3*[T] = { val x = *SRAM3*(s); x.p
Annotates that addresses in this SRAM2 can be read in parallel by factor **p**.

Used when creating references to sparse regions of DRAM.

def **load**(dram: *DRAM3*[T]): *Unit*
Create a dense, burst load from the given region of DRAM to this on-chip memory.

def **load**(dram: *DRAMDenseTile3*[T]): *Unit*
Create a dense, burst load from the given region of DRAM to this on-chip memory.

class **SRAM4**[T] extends SRAM[T]

def **dim0**: *Index*
Returns the first dimension of this SRAM4.

def **dim1**: *Index*
Returns the second dimension of this SRAM4.

def **dim2**: *Index*
Returns the third dimension of this SRAM4.

def **dim3**: *Index*
Returns the fourth dimension of this SRAM4.

def **size**: *Index*
Returns the total size of this SRAM4.

def **apply**(a: *Index*, b: *Index*, c: *Index*, d: *Index*): T
Returns the value in this SRAM4 at the 4-dimensional address **a**, **b**, **c**, **d**.

def **update**(a: *Index*, b: *Index*, c: *Index*, d: *Index*, data: T): *Unit*
Updates the value in this SRAM4 at the 4-dimensional address to **data**.

def **load**(dram: *DRAM4*[T]): *Unit*
Create a dense, burst load from the given region of DRAM to this on-chip memory.

def **load**(dram: *DRAMDenseTile4*[T]): *Unit*
Create a dense, burst load from the given region of DRAM to this on-chip memory.

class **SRAM5**[T] extends SRAM[T]
def **dim0**: *Index* Returns the first dimension of this SRAM5.
def **dim1**: *Index* Returns the second dimension of this SRAM5.
def **dim2**: *Index* Returns the third dimension of this SRAM5.
def **dim3**: *Index* Returns the fourth dimension of this SRAM5.
def **dim4**: *Index* Returns the fifth dimension of this SRAM5.
def **size**: *Index* Returns the total size of this SRAM5.
def **apply**(a: *Index*, b: *Index*, c: *Index*, d: *Index*, e: *Index*): T Returns the value in this SRAM5 at the 5-dimensional address **a**, **b**, **c**, **d**, **e**.
def **update**(a: *Index*, b: *Index*, c: *Index*, d: *Index*, e: *Index*, data: T): *Unit* Updates the value in this SRAM5 at the 5-dimensional address to **data**.
def **load**(dram: *DRAM5*[T]): *Unit* Create a dense, burst load from the given region of DRAM to this on-chip memory.
def **load**(dram: *DRAMDenseTile5*[T]): *Unit* Create a dense, burst load from the given region of DRAM to this on-chip memory.

1.2.5 Scheduling Directives

While much of Spatial's scheduling is automated, users wishing to fine tune or annotate their applications can do so using several scheduling directives when declaring loops and controllers.

Pipe

Pipe is the default scheduling directive in Spatial, and doesn't usually need to be specified. This directive tells the compiler that the stages of the controller can be overlapped in a pipelined fashion. If the controller contains other controllers within it, this means that these inner controllers will be executed using coarse-grained pipeline scheduling.

Static methods

object **Pipe**
def **apply**(func: => *Unit*): *Unit* Creates a Unit Pipe, akin to a Foreach with one iteration.
def *Fold*: *Fold* References the *Fold* object with pipelining specified as the scheduling directive.
def *Foreach*: *Foreach* References the *Foreach* object with pipelining specified as the scheduling directive.
def *MemFold*: *MemFold* References the *MemFold* object with pipelining specified as the scheduling directive.
def *MemReduce*: *MemReduce* References the *MemReduce* object with pipelining specified as the scheduling directive.
def *Reduce*: *Reduce* References the *Reduce* object with pipelining specified as the scheduling directive.

Sequential

Sequential is a scheduling directive which tells the compiler not to attempt to parallelize or to pipeline inner computation. In this scheduling mode, the controller's counter will only increment when it's last stage is complete. This directive is needed primarily when the algorithm contains long loop-carry dependencies that cannot be optimized away.

Static methods

object **Sequential**
def **apply**(func: => *Unit*): *Unit* Creates a Unit Pipe, akin to a Foreach with one iteration.
def *Fold*: *Fold* References the *Fold* object with sequential execution specified as the scheduling directive.
def *Foreach*: *Foreach* References the *Foreach* object with sequential execution specified as the scheduling directive.
def *MemFold*: *MemFold* References the *MemFold* object with sequential execution specified as the scheduling directive.
def *MemReduce*: *MemReduce* References the *MemReduce* object with sequential execution specified as the scheduling directive.
def *Reduce*: *Reduce* References the *Reduce* object with sequential execution specified as the scheduling directive.

Stream

Stream is a scheduling directive which tells the compiler to overlap inner computation in a fine-grained, streaming fashion. In controllers which contain multiple control stages, this implies that communication is being done through *FIFO* memories. at an element-wise level.

Communication across stages within Stream controllers through any memory except FIFOs is currently disallowed. Note that this may change as the language evolves.

Static methods

object **Stream**
def **apply**(func: => *Unit*): *Unit* Creates a streaming Unit Pipe, akin to a Foreach with one iteration.
def *Fold*: *Fold* References the *Fold* object with streaming specified as the scheduling directive.
def *Foreach*: *Foreach* References the *Foreach* object with streaming specified as the scheduling directive.
def *MemFold*: *MemFold* References the *MemFold* object with streaming specified as the scheduling directive.
def *MemReduce*: *MemReduce* References the *MemReduce* object with streaming specified as the scheduling directive.
def *Reduce*: *Reduce* References the *Reduce* object with streaming specified as the scheduling directive.

1.3 Host Code

This section lists types and methods which are non-acceleratable, i.e. only available on the CPU or in simulation.

1.3.1 Array

Class and companion object for managing one dimensional arrays on the CPU.

Note that this type shadows the unstaged Scala Array. In the case where an unstaged Array type is required, use the full *scala.Array* name.

Constructor

The following syntax is available for constructing Arrays from indexed functions:

```
(0::32){i => func(i) }
```

This returns an Array of size 32 with elements defined by *func(i)*. More general *Range* forms can also be used, including strided (e.g. 0::2::8) and offset (e.g. 32::64). The iterator *i* will iterate over all values in the supplied range.

Static methods

object **Array**
def **tabulate**[T:*Type*](size: *Index*)(func: *Index* => T): *Array*[T] Returns an immutable Array with the given **size** and elements defined by **func**.
def **
def **fill**[T:*Type*](size: *Index*)(func: => T): *Array*[T] = this.tabulate(size){ _ Returns an immutable Array with the given **size** and elements defined by **func**. Note that while **func** does not depend on the index, it is still executed **size** times.
def **empty**[T:*Type*](size: *Index*): *Array*[T] Returns an empty, mutable Array with the given **size**.
def **apply**[T:*Type*](elements: T*): *Array*[T] Returns an immutable Array with the given elements.

Infix methods

class **Array**[T]

def **length**: *Index*
Returns the size of this Array.

def **apply**(i: *Index*): T
Returns the element at index **i**.

def **update**[A](i: *Index*, data: A)(implicit lift: Lift[A,T]): *Unit*
Updates the element at index **i** to data.

def **foreach**(func: T => *Unit*): *Unit*
Applies the function **func** on each element in the Array.

def **map**[R:*Type*](func: T => R): *Array*[R]
Returns a new Array created using the mapping **func** over each element in this Array.

def **zip**[S:*Type*,R:*Type*](that: *Array*[S])(func: (T,S) => R): *Array*[R]
Returns a new Array created using the pairwise mapping **func** over each element in this Array and the corresponding element in **that**.

def **reduce**(rfunc: (T,T) => T): T
Reduces the elements in this Array into a single element using associative function **rfunc**.

def **fold**(init: T)(rfunc: (T,T) => T): T
Reduces the elements in this Array and the given initial value into a single element using associative function **rfunc**.

def **filter**(cond: T => MBoolean): *Array*[T]
Returns a new Array with all elements in this Array which satisfy the given predicate **cond**.

def **flatMap**[R:*Type*](func: T => *Array*[R]): *Array*[R]
Returns a new Array created by concatenating the results of **func** applied to all elements in this Array.

def **groupByReduce**[K:*Type*,V:*Type*](key: A => K)(value: A => V)(reduce: (V,V) => V): *HashMap*[K,V]
Partitions this Array using the **key** function, then maps each element using **value**, and finally combines values in each bin using the associative **reduce** function.

def **mkString**(delimeter: *String*)
Creates a string representation of this Array using the given **delimeter**.

def **mkString**(start: *String*, delimeter: *String*, stop: *String*): *String*
Creates a string representation of this Array using the given **delimeter**, bracketed by **start** and **stop**.

def **reshape**(rows: *Index*, cols: *Index*): *Matrix*[T]
Returns an immutable view of the data in this Array as a *Matrix* with given **rows** and **cols**.

def **reshape**(dim0: *Index*, dim1: *Index*, dim2: *Index*): *Tensor3*[T]
Returns an immutable view of the data in this Array as a *Tensor3* with given dimensions.

def **reshape**(dim0: *Index*, dim1: *Index*, dim2: *Index*, dim3: *Index*): *Tensor4*[T]
Returns an immutable view of the data in this Array as a *Tensor4* with given dimensions.

def **reshape**(dim0: *Index*, dim1: *Index*, dim2: *Index*, dim3: *Index*, dim4: *Index*): *Tensor5*[T]
Returns an immutable view of the data in this Array as a *Tensor5* with given dimensions.

def **!=**(that: *Array*[T]): MBoolean = this.zip(that){(x,y) => x =!

1.3.2 Debugging Operations

These operations are available for use on the CPU and during simulation to aid runtime debugging.

Methods

def **assert**(cond: MBoolean, msg: *String*): *Unit* Checks that the given condition **cond** is true at runtime. If not, exits the program with the given message.
def **assert**(cond: MBoolean): *Unit* Checks that the given condition **cond** is true at runtime. If not, exits the program with a generic exception.
def **println**(): *Unit* Prints an empty line to the console.
def **print**[T:*Type*](x: T): *Unit* Prints a String representation of the given value to the console.
def **println**[T:*Type*](x: T): *Unit* Prints a String representation of the given value to the console followed by a linebreak.
def **print**(x: *String*): *Unit* Prints the given String to the console.
def **println**(x: *String*): *Unit* Prints the given String to the console, followed by a linebreak.
def **printArray**[T:*Type*](array: *Array*[T], heading: *String* = “”): *Unit* Prints the given Array to the console, preceded by an optional heading.
def **printMatrix**[T:*Type*](matrix: *Matrix*[T], heading: *String* = “”): *Unit* Prints the given Matrix to the console, preceded by an optional heading.
def **printTensor3**[T:*Type*](tensor: *Tensor3*[T], heading: *String* = “”): *Unit* Prints the given Tensor3 to the console, preceded by an optional heading.
def **printTensor4**[T:*Type*](tensor: *Tensor4*[T], heading: *String* = “”): *Unit* Prints the given Tensor4 to the console, preceded by the an optional heading.
def **printTensor5**[T:*Type*](tensor: *Tensor5*[T], heading: *String* = “”): *Unit* Prints the given Tensor5 to the console, preceded by the an optional heading.

1.3.3 File I/O

File I/O operations are available for use on the host PU and during simulation.

Methods

def **loadCSV1D**[T:*Type*](filename: *String*, delimiter: *String*)(implicit cast: Cast[*String*,T]): *Array*[T] Loads the CSV at **filename** as an *Array* using the supplied **delimiter** for parsing. The delimiter defaults to a comma if none is supplied.
def **loadCSV2D**[T:*Type*](filename: *String*, delimiter: *String*)(implicit cast: Cast[*String*,T]): *Matrix*[T] Loads the CSV at **filename** as a *Matrix*, using the supplied element delimiter and linebreaks across rows.
def **writeCSV1D**[T:*Type*](array: *Array*[T], filename: *String*, delimiter: *String*): *Unit* Writes the given Array to the file at **filename** using the given **delimiter**. If no delimiter is given, defaults to comma.
def **writeCSV2D**[T:*Type*](matrix: *Matrix*[T], filename: *String*, delimiter: *String*): *Unit* Writes the given Matrix to the file at **filename** using the given element **delimiter**. If no element delimiter is given, defaults to comma.

Under development

The following methods are not yet fully supported. If you need support for these methods, please contact the Spatial group.

def **loadBinary**[T:*Type*:*Num*](filename: *String*): *Array*[T] Loads the given binary file at **filename** as an *Array*.
def **writeBinary**[T:*Type*:*Num*](array: *Array*[T], filename: *String*): *Unit* Saves the given Array to disk as a binary file at **filename**.

1.3.4 HashMap

Class and companion object for an immutable map on the host CPU.

Infix methods

class **HashMap**[K, V]
def **size**: *Int* Returns the number of key-value pairs stored in this HashMap.
def **keys**: *Array*[K] Returns an Array of all keys stored in this HashMap.
def **values**: *Array*[V] Returns an Array of all values stored in this HashMap.
def **apply**(key: K): V Returns the value associated with the given key. Throws an exception if the given key is not stored in this HashMap
def **contains**(key: K): *Boolean* Returns the value associated with the given key. Throws an exception if the given key is not stored in this HashMap

1.3.5 Matrix

Class and companion object for dense matrices on the CPU.

Constructor

The following syntax is available for constructing Matrix instances from indexed functions:

```
(0::16, 0::32){(i,j) => func(i,j) }
```

This returns a Matrix with 16 rows and 32 columns, with elements defined by *func(i,j)*. More general *Range* forms can also be used, including strided (e.g. 0::2::8) and offset (e.g. 32::64). The iterators *i* and *j* will iterate over all values in their respective ranges.

Static methods

object **Matrix**
def **tabulate**[T:*Type*](rows: *Index*, cols: *Index*)(func: (*Index*, *Index*) => T): *Matrix*[T] Returns an immutable Matrix with the given **rows** and **cols** and elements defined by **func**.
def **fill**[T:*Type*](rows: *Index*, cols: *Index*)(func: => T): *Matrix*[T] = this.tabulate(rows, cols){(_,_) Returns an immutable Matrix with the given **rows** and **cols** and elements defined by **func**. Note that while **func** does not depend on the index, it is still executed **rows*****cols** times.

Infix methods

class **Matrix**
def **rows**: *Index* Returns the number of rows in this Matrix.
def **cols**: *Index* Returns the number of columns in this Matrix.
def **apply**(i: *Index*, j: *Index*): T Returns the element at the given two-dimensional address (**i**, **j**).
def **update**(i: *Index*, j: *Index*, elem: T): *Unit* Updates the element at the given two dimensional address to **elem**.
def **flatten**: *Array*[T] Returns a flattened, immutable *Array* view of this Matrix's data.
def **foreach**(func: T => *Unit*): *Unit* Applies the function **func** on each element in this Matrix.
def **map**[R:*Type*](func: T => R): *Matrix*[R] Returns a new Matrix created using the mapping **func** over each element in this Matrix.
def **zip**[S:*Type*,R:*Type*](that: *Matrix*[S])(func: (T,S) => R): *Matrix*[R] Returns a new Matrix created using the pairwise mapping **func** over each element in this Matrix and the corresponding element in **that**.
def **reduce**(rfunc: (T,T) => T): T Reduces the elements in this Matrix into a single element using associative function **rfunc**.
def **transpose**(): *Matrix*[T] = (0::cols, 0::rows){(j, i) Returns the transpose of this Matrix.

1.3.6 String

Staged type representing arbitrary length Strings. Note that this type shadows the respective unstaged Scala type. In the case where an unstaged type is required, use the full name *java.lang.String*.

Infix methods

class **String**

def **to**[T]: T

Converts this String to the specified type

[**NOTE**] This method is currently defined for:

- *FixPt[_,_,_]*
- *FltPt[_,_]*
- *Boolean*

1.3.7 Tensor

Tensors are multi-dimension, dense arrays with more than 2 dimensions. Currently 3 - 5 dimension Tensors are supported.

Like *Array* and *Matrix*, Tensors can only be used in host code. In accelerator code, use *DRAM* (for off-chip) or *SRAM* (on-chip) memories for multi-dimensional array support.

Constructors

Spatial includes syntax for constructing Tensor instances from indexed functions.

The following returns a 16 x 32 x 8 Tensor3, with elements defined by *func(i,j,k)*:

```
(0::16, 0::32, 0::8){(i,j,k) => func(i,j,k) }
```

A Tensor4 can be constructed in a similar way:

```
(0::4, 0::16, 0::8, 0::32){(a,b,c,d) => func(a,b,c,d) }
```

As can a Tensor5:

```
(0::2, 0::4, 0::5, 0::3, 0::32){(a,b,c,d,e) => func(a,b,c,d,e) }
```

More general *Range* forms can also be used, including strided (e.g. 0::2::8) and offset (e.g. 32::64). Iterators (e.g. *i*, *j*, *k* in the above examples) will iterate over all values in their respective ranges.

Static methods

object **Tensor3**

def **tabulate**[T:*Type*](dim0: *Index*, dim1: *Index*, dim2: *Index*)(func: (*Index*, *Index*, *Index*) => T): *Tensor3*[T]

Returns an immutable Tensor3 with the given dimensions and elements defined by **func**.

def **fill**[T:*Type*](dim0: *Index*, dim1: *Index*, dim2: *Index*)(func: => T): *Tensor3*[T]

Returns an immutable Tensor3 with the given dimensions and elements defined by **func**.

Note that while **func** does not depend on the index, it is still executed multiple times.

object **Tensor4**

def **tabulate**[T:*Type*](dim0: *Index*, dim1: *Index*, dim2: *Index*, dim3: *Index*)(func: (*Index*, *Index*, *Index*, *Index*) => T): *Tensor4*[T]

Returns an immutable Tensor4 with the given dimensions and elements defined by **func**.

def **fill**[T:*Type*](dim0: *Index*, dim1: *Index*, dim2: *Index*, dim3: *Index*)(func: => T): *Tensor4*[T]

Returns an immutable Tensor4 with the given dimensions and elements defined by **func**.

Note that while **func** does not depend on the index, it is still executed multiple times.

object **Tensor5**

def **tabulate**[T:*Type*](dim0: *Index*, dim1: *Index*, dim2: *Index*, dim3: *Index*, dim4: *Index*)(func: (*Index*, *Index*, *Index*, *Index*, *Index*) => T): *Tensor5*[T]

Returns an immutable Tensor5 with the given dimensions and elements defined by **func**.

def **fill**[T:*Type*](dim0: *Index*, dim1: *Index*, dim2: *Index*, dim3: *Index*, dim4: *Index*)(func: => T): *Tensor5*[T]

Returns an immutable Tensor5 with the given dimensions and elements defined by **func**.

Note that while **func** does not depend on the index, it is still executed multiple times.

Infix methods

class **Tensor3**[T]

def **dim0**: *Index*

Returns the first dimension of this Tensor3.

def **dim1**: *Index*

Returns the second dimension of this Tensor3.

def **dim2**: *Index*

Returns the third dimension of this Tensor3.

def **apply**(i: *Index*, j: *Index*, k: *Index*): T

Returns the element in this Tensor3 at the given 3-dimensional address.

def **update**(i: *Index*, j: *Index*, k: *Index*, elem: T): *Unit*

Updates the element at the given 3-dimensional address to **elem**.

def **flatten**: *Array*[T]

Returns a flattened, immutable *Array* view of this Tensor3's data.

def **foreach**(func: T => *Unit*): *Unit*

Applies the function **func** on each element in this Tensor3.

def **map**[R:*Type*](func: T => R): *Tensor3*[R]

Returns a new Tensor3 created using the mapping **func** over each element in this Tensor3.

def **zip**[S,R:*Type*](that: *Tensor3*[S])(func: (T,S) => R): *Tensor3*[R]

Returns a new Tensor3 created using the pairwise mapping **func** over each element in this Tensor3 and the corresponding element in **that**.

def **reduce**(rfunc: (T,T) => T): T

Reduces the elements in this Tensor3 into a single element using associative function **rfunc**.

class **Tensor4**[T]

def **dim0**: *Index*
Returns the first dimension of this Tensor4.

def **dim1**: *Index*
Returns the second dimension of this Tensor4.

def **dim2**: *Index*
Returns the third dimension of this Tensor4.

def **dim3**: *Index*
Returns the fourth dimension of this Tensor4.

def **apply**(i: *Index*, j: *Index*, k: *Index*, l: *Index*): T
Returns the element in this Tensor4 at the given 4-dimensional address.

def **update**(i: *Index*, j: *Index*, k: *Index*, l: *Index*, elem: T): *Unit*
Updates the element at the given 4-dimensional address to **elem**.

def **flatten**: *Array*[T]
Returns a flattened, immutable *Array* view of this Tensor4's data.

def **foreach**(func: T => *Unit*): *Unit*
Applies the function **func** on each element in this Tensor4.

def **map**[R:*Type*](func: T => R): *Tensor4*[R]
Returns a new Tensor4 created using the mapping **func** over each element in this Tensor4.

def **zip**[S,R:*Type*](b: *Tensor4*[S])(func: (T,S) => R): *Tensor4*[R]
Returns a new Tensor4 created using the pairwise mapping **func** over each element in this Tensor4 and the corresponding element in **that**.

def **reduce**(rfunc: (T,T) => T): T
Reduces the elements in this Tensor4 into a single element using associative function **rfunc**.

class **Tensor5**[T]
def **dim0**: *Index* Returns the first dimension of this Tensor5.
def **dim1**: *Index* Returns the second dimension of this Tensor5.
def **dim2**: *Index* Returns the third dimension of this Tensor5.
def **dim3**: *Index* Returns the fourth dimension of this Tensor5.
def **dim4**: *Index* Returns the fifth dimension of this Tensor5.
def **apply**(i: *Index*, j: *Index*, k: *Index*, l: *Index*, m: *Index*): T Returns the element in this Tensor5 at the given 5-dimensional addreess.
def **update**(i: *Index*, j: *Index*, k: *Index*, l: *Index*, m: *Index*, elem: T): *Unit* Updates the element at the given 5-dimensional address to **elem**.
def **flatten**: *Array*[T] Returns a flattened, immutable *Array* view of this Tensor5's data.
def **foreach**(func: T => *Unit*): *Unit* Applies the function **func** on each element in this Tensor5.
def **map**[R:*Type*](func: T => R): *Tensor5*[R] Returns a new Tensor5 created using the mapping **func** over each element in this Tensor5.
def **zip**[S,R:*Type*](b: *Tensor5*[S])(func: (T,S) => R): *Tensor5*[R] Returns a new Tensor5 created using the pairwise mapping **func** over each element in this Tensor5 and the corresponding element in **that**.
def **reduce**(rfunc: (T,T) => T): T Reduces the elements in this Tensor5 into a single element using associative function **rfunc**.

1.3.8 Transfer Operations

These operations are used to transfer scalar values and arrays between the CPU host and the hardware accelerator. They must be specified explicitly in the host code (not in Accel scopes).

Methods

def **setArg**[T](reg: Reg[T], value: T): *Unit* Transfer a scalar value from the host to the accelerator through the register **reg**. **reg** should be allocated as the HostIO or ArgIn methods.
def **getArg**[T:*Type*:*Bits*](reg: Reg[T]): T Transfer a scalar value from the accelerator to the host through the register **reg**. **reg** should be allocated using the HostIO or ArgIn methods.
def **setMem**[T:*Type*:*Bits*](dram: *DRAM*[T], data: *Array*[T]): *Unit* Transfers the given *Array* of **data** from the host's memory to **dram**'s region of accelerator DRAM.
def **getMem**[T:*Type*:*Bits*](dram: *DRAM*[T]): *Array*[T] Transfers **dram**'s region of accelerator DRAM to the host's memory and returns the result as an *Array*.
def **setMem**[T:*Type*:*Bits*](dram: *DRAM*[T], data: *Matrix*[T]): *Unit* Transfers the given *Matrix* of **data** from the host's memory to **dram**'s region of accelerator DRAM.
def **getMatrix**[T:*Type*:*Bits*](dram: *DRAM2*[T])(implicit ctx: SrcCtx): *Matrix*[T] Transfers **dram**'s region of accelerator DRAM to the host's memory and returns the result as a *Matrix*.
def **setMem**[T:*Type*:*Bits*](dram: *DRAM*[T], tensor3: *Tensor3*[T]): *Unit* Transfers the given Tensor3 of **data** from the host's memory to **dram**'s region of accelerator DRAM.
def **getTensor3**[T:*Type*:*Bits*](dram: *DRAM3*[T])(implicit ctx: SrcCtx): *Tensor3*[T] Transfers **dram**'s region of accelerator DRAM to the host's memory and returns the result as a Tensor3.
def **setMem**[T:*Type*:*Bits*](dram: *DRAM*[T], tensor4: *Tensor4*[T]): *Unit* Transfers the given Tensor4 of **data** from the host's memory to **dram**'s region of accelerator DRAM.
def **getTensor4**[T:*Type*:*Bits*](dram: *DRAM4*[T])(implicit ctx: SrcCtx): *Tensor4*[T] Transfers **dram**'s region of accelerator DRAM to the host's memory and returns the result as a Tensor4.
def **setMem**[T:*Type*:*Bits*](dram: *DRAM*[T], tensor5: *Tensor5*[T]): *Unit* Transfers the given Tensor5 of **data** from the host's memory to **dram**'s region of accelerator DRAM.
def **getTensor5**[T:*Type*:*Bits*](dram: *DRAM5*[T])(implicit ctx: SrcCtx): *Tensor5*[T] Transfers **dram**'s region of accelerator DRAM to the host's memory and returns the result as a Tensor5.

1.4 The Spatial Compiler

1.4.1 Compilation

The Spatial Compiler is run using:

```
bin/spatial <Application Name> [flags]
```

where <*Application Name*> is the name of the Spatial application object.

1.4.2 Compiler Flags

The following compiler flags are available in the Spatial compiler:

–sim	Turns on the Scala backend for functional simulation.
–synth	Turns on the Chisel RTL backend for cycle-accurate simulation and/or synthesis.
–instrument	Enables RTL instrumentation hooks for manual pipeline balancing analyses
–retime	Enables RTL retiming to meet higher clock speeds.
	This flag will eventually be enabled by default.
–syncMem	Enables synchronous memory operations for all SRAMs to use fewer resources. Also enables *–retime*.
–out <dir>	Specifies an output directory to place generated code. Default directory is *gen/<app name>*.

1.4.3 Advanced Compiler Flags

These flags are primarily for use in development of the Spatial compiler.

–cheapFifos	Uses "cheap" FIFOs if there are no FIFOs with lane-dependent enqueues or dequeues.
–multifile	0-6, default = 4
–naming	Turns on useful naming for assistance in debugging generated chisel
–dse	Run compiler design space exploration. Interactivity is planned, but currently just generates a data file.

More to be added. . .

CHAPTER 2

Tutorial

This section provides a user's introduction to Spatial to help you get started using the language. Feel free to skip around using this index or to follow the tutorial step by step. The tutorial is structured as a sequence of applications, each demonstrating additional complexity and APIs available in Spatial.

2.1 0. Getting Started

2.1.1 Prerequisites

First, make sure to download and install the following prerequisites:

- Scala SBT
- Java JDK
- VCS ** NOTE: License is required **

While it's not at all required, it may be easier to learn to use Spatial if you've had experience with Scala or a similar functional programming language in the past. Knowledge of Scala will allow you to use meta-programming to assist your Spatial designs.

If you'd like, check out this Scala tutorial .

Finally, please sign up for the Spatial users google group if you have any questions.

2.1.2 Installation (From Source)

Run the following (bash) commands to clone and update the spatial-lang repository:

```
git clone https://github.com/stanford-ppl/spatial-lang.git
cd spatial-lang
git submodule update --init
```

This will pull Spatial's submodules *argon*, *apps*, and *scala-virtualized*.

You may need to export your JAVA_HOME environment variable to point to your Java installation (usually /usr/bin)

You are now ready to compile the language. Run the following:

```
cd spatial-lang # Navigate to root of spatial-lang repository
sbt compile
```

A good habit would be to pull from these repositories often and run `sbt compile` in your spatial-lang directory.

That's it! Up next, you will learn how to use the language by working through a series of examples. The concepts you will learn in these tutorials are listed below. Feel free to skip around the apps as you find convenient:

- *Hello, World!*
- Application skeleton (import statements, application creation, accel scope, host scope)
- ArgIn
- ArgOut
- HostIO
- DRAM
- SRAM
- Reg
- Typing system
- Data transfer between host and accel (setArg, setMem, getArg, getMem, load, store, gather, scatter)
- Basic debugging hooks
- Compiling an app
- *Dot Product*
- Tiling
- Reduce and Fold
- Sequential execution and Coarse-grain pipelining
- Parallelization
- Basic buffering and banking
- *General Matrix Multiply (GEMM)*
- MemReduce and MemFold
- Debugging with instrumentation
- Advanced banking
- Advanced buffering
- *Differentiator & Sobel Filter*
- LineBuffer
- ShiftRegister
- LUT
- Spatial Functions and Multifile Projects

- *Needleman-Wunsch*
- FSM
- Branching
- FIFO
- Systolic Arrays
- File IO and text management
- Asserts, Breakpoints, and Sleep

2.2 1. Hello, Spatial!

2.2.1 Catalog of Features

In this section, you will learn about the following components in Spatial:

- Application skeleton (import statements, application creation, accel scope, host scope)
- DRAM
- SRAM
- ArgIn
- ArgOut
- HostIO
- Reg
- Typing system
- Data transfer between host and accel (setArg, setMem, getArg, getMem, load, store, gather, scatter)
- Basic debugging hooks
- Compiling an app

2.2.2 Application Overview

In this section, you will see how to put together the bare-minimum Spatial application. While the code does not do any "meaningful" work, it demonstrates the basic primitives that almost all applications have and is intended to be the "Hello, world!" program for hardware. You will start by generating input and output registers to get the accelerator and host to interact with each other, and then add tile transfers between the off-chip DRAM and on-chip SRAM. You will then learn what functions are provided to test functionality and utilize the host. Finally, you will learn the basic compilation flows for testing the functionality of the algorithm, cycle-accurate simulation of the generated RTL, and bitstream generation to deploy to a supported FPGA or architecture.

Below is a visualization of what we will be doing in this tutorial. We start with a host and an FPGA, both connected to DRAM. We will then instantiate all of the different ways you can get the two processors to interact with each other. We will create an RTL that will sit inside the FPGA, as well as some C++ code that will sit inside the host. Spatial automatically instantiates a box called "Fringe," which is an FPGA-agnostic hardware design that allows the RTL to interact with peripherals, DRAM, PCIe buses, and whatever else is available on a given SoC or FPGA board.

2.2.3 Application Template

All Spatial programs have a few basic components. The following code example shows each of those components for an application that is called *HelloSpatial*:

```
import spatial.dsl._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main() {


    Accel {

    }

  }
}
```

2.2.4 Compiling

We will use the above template to learn the process for compiling, simulating, and synthesizing a design. While this template is empty, you can use this same flow freely as you build your applications in the tutorials below.

Currently, you should edit and place apps inside of your *spatial-lang/apps/src/* directory. Copy-paste the above template into a new file in this directory and you are ready to compile.

There are currently two backend targets that you can compile to: Scala and RTL.

The Scala backend is the fastest way to check the correctness of your application. It does not simulate parallelization and pipelining as they would execute in hardware so any race conditions or loop-carry dependency issues will not be caught in this backend, but it is a very useful tool for early and rapid development.

The RTL backend can be both simulated on a cycle-accurate simulator and synthesized to generate a bitstream for an FPGA. The RTL simulation is the most accurate way to test your design, but takes on the order of minutes to compile.

Compiling to Scala

Targetting Scala is the quickest way to simulate your app and test for basic functional correctness. You should use this backend if you are debugging things at the algorithm level. In order to compile and simulate for the Scala backend, run:

```
cd spatial-lang/ # Navigate to Spatial base directory
bin/spatial <app name> --sim # + :doc:`other options <../compiler>`
```

The "<app name>" refers to the name of the `object`. In our app above, for example, the app name is "HelloSpatial". See the "Running" section below for a guide on how to test the generated app

Compiling to RTL

Targeting Chisel will let you compile your app down into Berkeley's Chisel language, which eventually compiles down to Verilog. It also allows you to debug your app at the clock-cycle resolution. In order to compile with the Chisel backend, run the following:

```
cd spatial-lang/ # Navigate to Spatial base directory
bin/spatial <app name> --synth # + :doc:`other options <../compiler>`
```

2.2.5 Synthesizing and Testing

After you have used the `bin/spatial` script to compile the app, navigate to the generated code directory to test the app. By default, this is `spatial-lang/gen/<app name>`. You will see some files and directories in this folder that correspond to the code that Spatial created for the various target platforms. For the RTL backend, here is a rough breakdown of what the important files are:

chisel/RootController.scala	Main trait where all of the controller and dataflow connections are made
chisel/x###.scala	Nested traits where more controller and dataflow connections are made
chisel/IOModule.scala	Interface between FPGA accelerator and CPU
chisel/BufferControlCxns	Connections for all N-buffered memories in the design
chisel/resources/*.scala	Files for all of the fundamental building blocks of a Spatial app
cpp/TopHost.scala	Contains the Application method where all CPU code is generated
controller_tree.html	Helpful diagram for showing the hierarchy of control nodes in your app

In order to finally test this code, you must compile the backend code itself. In order to do so, run the following:

```
cd gen/<app name>

# Choose ONE of the following
make sim # If you chose the Scala backend
make vcs # Cycle-accurate RTL simulation
make aws-F1 # Synthesize for Amazon F1
make zynq # Synthesize for Xilinx Zynq ZC706 or ZC702
make zcu # Synthesize for Xilinx ZCU102
make de1soc # Synthesize for Altera DE1SoC

# Run simulation executable if one of the first two options were chosen
bash run.sh "<arguments>"
```

NOTE: The "<arguments>" should be a space-separated list, fully enclosed in quotes. For example, an app that takes arguments 192 96 should be run with:

```
bash run.sh "192 96"
```

If you've forgotten what the command line arguments are for this app, you can always run:

```
bash run.sh --help
```

After running an RTL simulation, you can see the waveforms generated in the `test_run_dir/app.Launcher####` folder, with the *.vcd* extension for further debugging

** Synthesized bitstream process TBA **

2.2.6 DRAM Transfers

We will now continue developing a Spatial app based on the above skeleton. Please see the end of this section for a complete, copy-paste version of the code outlined below.

We will now add the code that will allow us to **1)** create data inside the host, **2)** transfer this data to DRAM where it can be acessed by the FPGA, **3)** load the data, **4)** interact with the data in on-chip SRAM, and **5)** store the data back to DRAM where it can be accessed by the host.

First, let's create a few data structures inside *main*, above the *Accel* block:

```
val data1D        = Array.tabulate(64){i => i * 3} // Create 1D array with 64 elements, each␣
↪element being index * 3
val data1D_longer = Array.tabulate(1024){i => i} // Create 1D array with 1024 elements
```

```
val data2D        = (0::64, 0::64){(i,j) => i*100 + j} // Create 64x64 2D, where each element is␣
↪row * 100 + col
val data5D        = (0::2, 0::2, 0::2, 0::2, 0::16){(i,j,k,l,m) => random[Int](5)} // Create 5D␣
↪tensor, the highest dimension tensor currently supported in Spatial, with each element a␣
↪random Int between 0 and 5
```

Now, let's allocate space in DRAM to memcpy this data to, so that the FPGA can read it later. This code also lives above the *Accel* block:

```
val dram1D        = DRAM[Int](64)
val dram1D_longer = DRAM[Int](1024)
val dram2D        = DRAM[Int](64,64)
val dram5D        = DRAM[Int](2,2,2,2,16)
```

Next, we can transfer our generated data into these DRAM allocations, still above the *Accel* block:

```
setMem(dram1D, data1D)
setMem(dram1D_longer, data1D_longer)
setMem(dram2D, data2D)
setMem(dram5D, data5D)
```

We can also create a few DRAMs that will be written to by the Accel:

```
val dram_result2D = DRAM[Int](32,32)
val dram_scatter1D = DRAM[Int](1024)
```

Now, we will move into the *Accel* block to create some SRAMs to catch and hold data on-chip:

```
val sram1D        = SRAM[Int](64)
val sram2D        = SRAM[Int](32,32)
val sram5D        = SRAM[Int](2,2,2,2,16)
```

With these SRAMs declared, we can load data into them. DRAM is burst-addressable, relatively slow memory. The Fringe module manages the command and data streams that connect the FPGA to DRAM:

```
sram1D load dram1D // Load data from a DRAM of matching dimension
sram2D load dram2D(32::64, 0::32 par 16) // Load region from DRAM. In this case, we load the␣
↪bottom-left quadrant of data from dram2D
sram5D load dram5D // Load 5D tensor
```

In the above snippet, notice that you can parallelize these operations. Parallelization of the leading dimension of the load into the 2D SRAM means that rather than funnelling a 512-bit burst (consisting of 16 ints that are 32 bits each) into 1 element at a time, we can store 16 elements at a time (the entire burst) into SRAM at once with each incoming burst.

Storing data from SRAM back into DRAM is straightforward, and can also have parallelization:

```
dram_result2D(0::32, 0::32 par 8) store sram2D
```

The Fringe module also makes it very straightfoward to do scatter and gather operations from DRAM. Because DRAM is burst-addressable, it can be very inefficient to interact with individual, non-consecutive addresses. The scatter and gather templates instantiate the control logic, caches, and other support required to efficiently coalesce, manage, and interact with DRAM at the word level.

> val gathered_sram = SRAM[Int](64) // Create SRAM to hold data gathered_sram gather dram1D_longer(sram1D par 1, 64) // Use the first 64 elements in sram1D as the addresses in dram1D_longer to collect, and store them into gathered_sram

We can also scatter this data back into DRAM

> dram_scatter1D(sram1D par 1, 64) scatter gathered_sram // For the first 64 elements, place element i of gathered_sram into the address indicated by the i'th element of sram1D

Now, let's move outside the Accel and load our data back into the host to check if it is correct:

```
val result_scattered = getMem(dram_scatter1D)
val result2D = getMatrix(dram_result2D) // Collect 2D dram as a "Matrix."  Likewise, 3, 4, and␣
↪5D regions use "getTensor3", "getTensor4", and "getTensor5"
```

Finally, let's check if the data is correct and print the results. Note that while print lines inside the host code will print for both the Scala and RTL backends, print lines inside the Accel will only print in the Scala backend and will be ignored in RTL, since there is no straightforward print for FPGAs:

```
printMatrix(result2D, "Result 2D: ") // printTensor3, printTensor4, and printTensor5 also exist
printArray(result_scattered, "Result Scattered: ")
val gold_2D = (32::64, 0::32){(i,j) => i*100 + j} // Remember we took bottom-left corner
val cksum_2D = gold_2D.zip(result2D){_==_}.reduce{_&&_} // Zip the gold with the result and␣
↪check if they are all equal
val cksum_scattered = Array.tabulate(64){i => result_scattered(3*i) == 3*i}.reduce{_&&_} //␣
↪Check if every 3 entries is equal to the index
println("2D pass? " + cksum_2D)
println("scatter pass? " + cksum_scattered)
```

Congratulations! You have completed the DRAM section of the tutorial. Please reference the *Compiling* and *Synthesizing and Testing* sections above for a refresher on how to test your app.

Below is a copy-pastable version of the code outlined above:

```
import spatial.dsl._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main() {

    val data1D        = Array.tabulate(64){i => i * 3} // Create 1D array with 64 elements, each␣
↪element being index * 3
    val data1D_longer = Array.tabulate(1024){i => i} // Create 1D array with 1024 elements
    val data2D        = (0::64, 0::64){(i,j) => i*100 + j} // Create 64x64 2D, where each␣
↪element is row * 100 + col
    val data5D        = (0::2, 0::2, 0::2, 0::2, 0::16){(i,j,k,l,m) => random[Int](5)} // Create␣
↪5D tensor, the highest dimension tensor currently supported in Spatial, with each element a␣
↪random Int between 0 and 5

    val dram1D        = DRAM[Int](64)
    val dram1D_longer = DRAM[Int](1024)
    val dram2D        = DRAM[Int](64,64)
    val dram5D        = DRAM[Int](2,2,2,2,16)

    setMem(dram1D, data1D)
    setMem(dram1D_longer, data1D_longer)
    setMem(dram2D, data2D)
    setMem(dram5D, data5D)

    val dram_result2D = DRAM[Int](32,32)
    val dram_scatter1D = DRAM[Int](1024)

    Accel {
      val sram1D        = SRAM[Int](64)
      val sram2D        = SRAM[Int](32,32)
      val sram5D        = SRAM[Int](2,2,2,2,16)

      sram1D load dram1D // Load data from a DRAM of matching dimension
      sram2D load dram2D(32::64, 0::32 par 16) // Load region from DRAM. In this case, we load␣
↪the bottom-left quadrant of data from dram2D
      sram5D load dram5D // Load 5D tensor

      dram_result2D(0::32, 0::32 par 8) store sram2D
```

```
      val gathered_sram = SRAM[Int](64)  // Create SRAM to hold data
      gathered_sram gather dram1D_longer(sram1D par 1, 64)  // Use the first 64 elements in␣
↪sram1D as the addresses in dram1D_longer to collect, and store them into gathered_sram

      dram_scatter1D(sram1D par 1, 64) scatter gathered_sram // For the first 64 elements, place␣
↪element i of gathered_sram into the address indicated by the i'th element of sram1D
    }

    val result_scattered = getMem(dram_scatter1D)
    val result2D = getMatrix(dram_result2D) // Collect 2D dram as a "Matrix."  Likewise, 3, 4,␣
↪and 5D regions use "getTensor3D", "getTensor4D", and "getTensor5D"

    printMatrix(result2D, "Result 2D: ")
    printArray(result_scattered, "Result Scattered: ")
    val gold_2D = (32::64, 0::32){(i,j) => i*100 + j} // Remember we took bottom-left corner
    val cksum_2D = gold_2D.zip(result2D){_==_}.reduce{_&&_} // Zip the gold with the result and␣
↪check if they are all equal
    val cksum_scattered = Array.tabulate(64){i => result_scattered(3*i) == 3*i}.reduce{_&&_} //␣
↪Check if every 3 entries is equal to the index
    println("2D pass? " + cksum_2D)
    println("scatter pass? " + cksum_scattered)
  }
}
```

2.2.7 ArgIn/Out Interfaces and Typing

We will now continue developing our Spatial app above and add ArgIns, ArgOuts, HostIOs, and Regs.

While most data that people want to process reside inside of DRAM data structures, there are times when you may want to pass individual arguments between the Accel and the host. Some examples include passing parameters to the Accel, such as a damping factor in an algorithm like PageRank or data structure dimensions in an algorithm like GEMM, as well as passing parameters to the host in algorithms like Dot Product. Let us define a few of these registers above the Accel block inside the `main()` function:

```
val argin1 = ArgIn[Int]   // Register that is written to by the host and read from by the Accel
val argout1 = ArgOut[Int] // Register that is written to by the Accel and read from by the host
val io1 = HostIO[Int]     // Register that can be both written to and read from by the Accel and␣
↪the host
```

By this point, you have probably noticed that we keep specifying everything as an Int in square brackets. These square brackets are how Scala passes along type arguments. Spatial is a hardware language that supports a few types besides 32-bit integers and you can define them as follows:

```
type T = FixPt[FALSE, _16, _16] // 32-bit unsigned integer with 16 whole bits and 16 fractional␣
↪bits.
type Flt = Float // 32-bit standard Float
```

Now we can make another argument using the T type:

```
val argin2 = ArgIn[T]
```

Now that we have created these registers, we can load values into them:

```
setArg(argin1, args(0).to[Int]) // Set argument with the first command-line value
setArg(argin2, 7.to[T]) // Args do not necessarily need to be set with command-line values
setArg(io1, args(1).to[Int])
```

Let's move into the Accel and interact with these registers:

```
val reg1 = Reg[Int](5) // Create register with initial value of 5
val reg2 = Reg[T] // Default initial value for a Reg is 0
Pipe{reg1 := argin1} // Load from ArgIn
Pipe{reg2 := argin2} // Load from ArgIn
argout1 := reg1 + reg2.value.to[Int] // Cast the value in reg2 to Int and add it to reg1
io1 := reg1
```

In the snippet above, you may notice that there are two Pipes. This is the first example of where the user must be aware of the hardware to understand what logic is actually getting generated. The compiler scopes code into separate Blocks. Before this point, we have not scoped any code into anything other than the base, global block, meaning all of the hardware we generate will fire at the same time. In this particular example, we want `reg1` and `reg2` to be loaded before we sum them up, and therefore we should scope them out with `Pipe` in order to ensure the top-level controller will execute them one after another. Note that if retiming is turned on (see *compiler flags*), then we would not need to scope these operations out because all primitives inside of a block are retimed appropriately to ensure their values arrive as dictated by the code. Without retiming, however, all primitives can happen simultaneously and give an incorrect result. Later sections will discuss retiming and controller hierarchies further.

Now we can move outside the Accel and read the arg values:

```
val result1 = getArg(argout1)
val result2 = getArg(io1)

println("Received " + result1 + " and " + result2)
val cksum = (result1 == {args(0).to[Int] + args(1).to[Int]}) && (result2 == args(0).to[Int]) //␣
↪The {} brackets are Scala's way of scoping operations
println("ArgTest pass? " + cksum)
```

Congratulations! You have completed the ArgIn/Out section of the tutorial. Please reference the *Compiling* and *Synthesizing and Testing* sections above for a refresher on how to test your app.

2.2.8 Final Code

Below is a copy-pastable version of the code outlined above:

```
import spatial.dsl._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main() {

    val argin1 = ArgIn[Int]   // Register that is written to by the host and read from by the␣
↪Accel
    val argout1 = ArgOut[Int] // Register that is written to by the Accel and read from by the␣
↪host
    val io1 = HostIO[Int]     // Register that can be both written to and read from by the Accel␣
↪and the host

    type T = FixPt[FALSE, _16, _16] // 32-bit unsigned integer with 16 whole bits and 16␣
↪fractional bits.
    type Flt = Float // 32-bit standard Float

    val argin2 = ArgIn[T]

    setArg(argin1, args(0).to[Int]) // Set argument with the first command-line value
    setArg(argin2, 7.to[T]) // Args do not necessarily need to be set with command-line values
    setArg(io1, args(1).to[Int])

    val data1D        = Array.tabulate(64){i => i * 3} // Create 1D array with 64 elements, each␣
↪element being index * 3
    val data1D_longer = Array.tabulate(1024){i => i} // Create 1D array with 1024 elements
```

```
    val data2D        = (0::64, 0::64){(i,j) => i*100 + j} // Create 64x64 2D, where each␣
↪element is row * 100 + col
    val data5D        = (0::2, 0::2, 0::2, 0::2, 0::16){(i,j,k,l,m) => random[Int](5)} // Create␣
↪5D tensor, the highest dimension tensor currently supported in Spatial, with each element a␣
↪random Int between 0 and 5

    val dram1D        = DRAM[Int](64)
    val dram1D_longer = DRAM[Int](1024)
    val dram2D        = DRAM[Int](64,64)
    val dram5D        = DRAM[Int](2,2,2,2,16)

    setMem(dram1D, data1D)
    setMem(dram1D_longer, data1D_longer)
    setMem(dram2D, data2D)
    setMem(dram5D, data5D)

    val dram_result2D = DRAM[Int](32,32)
    val dram_scatter1D = DRAM[Int](1024)

    Accel {
      val sram1D        = SRAM[Int](64)
      val sram2D        = SRAM[Int](32,32)
      val sram5D        = SRAM[Int](2,2,2,2,16)

      sram1D load dram1D // Load data from a DRAM of matching dimension
      sram2D load dram2D(32::64, 0::32 par 16) // Load region from DRAM. In this case, we load␣
↪the bottom-left quadrant of data from dram2D
      sram5D load dram5D // Load 5D tensor

      dram_result2D(0::32, 0::32 par 8) store sram2D

      val gathered_sram = SRAM[Int](64)  // Create SRAM to hold data
      gathered_sram gather dram1D_longer(sram1D par 1, 64)  // Use the first 64 elements in␣
↪sram1D as the addresses in dram1D_longer to collect, and store them into gathered_sram

      dram_scatter1D(sram1D par 1, 64) scatter gathered_sram // For the first 64 elements, place␣
↪element i of gathered_sram into the address indicated by the i-th element of sram1D

      val reg1 = Reg[Int](5) // Create register with initial value of 5
      val reg2 = Reg[T] // Default initial value for a Reg is 0
      Pipe{reg1 := argin1} // Load from ArgIn
      Pipe{reg2 := argin2} // Load from ArgIn
      argout1 := reg1 + reg2.value.to[Int] // Cast the value in reg2 to Int and add it to reg1
      io1 := reg1
    }

    val result_scattered = getMem(dram_scatter1D)
    val result2D = getMatrix(dram_result2D) // Collect 2D dram as a "Matrix."  Likewise, 3, 4,␣
↪and 5D regions use "getTensor3D", "getTensor4D", and "getTensor5D"

    printMatrix(result2D, "Result 2D: ")
    printArray(result_scattered, "Result Scattered: ")
    val gold_2D = (32::64, 0::32){(i,j) => i*100 + j} // Remember we took bottom-left corner
    val cksum_2D = gold_2D.zip(result2D){_==_}.reduce{_&&_} // Zip the gold with the result and␣
↪check if they are all equal
    val cksum_scattered = Array.tabulate(64){i => result_scattered(3*i) == 3*i}.reduce{_&&_} //␣
↪Check if every 3 entries is equal to the index
    println("2D pass? " + cksum_2D)
    println("scatter pass? " + cksum_scattered)


    val result1 = getArg(argout1)
    val result2 = getArg(io1)

    println("Received " + result1 + " and " + result2)
    val cksum = (result1 == {args(0).to[Int] + args(1).to[Int]}) && (result2 == args(0).to[Int])␣
↪// The {} brackets are Scala's way of scoping operations
```

```
    println("ArgTest pass? " + cksum)

  }
}
```

2.2.9 Stream Interfaces

** This section is still under construction **

Finally, you will see how to create stream interfaces with peripheral devices that your FPGA may have access to. Generally, these involve LEDs, switches, buttons, GPIO pins, ADC streams, and sensor interfaces. A stream interface looks like exposed signal pins inside the FPGA and there may or may not be ready/valid signals routed alongside them. For example, switches are input streams that are always valid and LEDs are output streams that are always ready. A pixel buffer that may come with an ADC stream will likely have a *valid* signal to indicate to the Accel that there is data ready to be dequeued, and the FPGA would need to send back a *ready* signal to indicate that it is ready to receive and process new data.

These protocols are abstracted away by the compiler and all the user needs to do is instantiate the interfaces and use them in the code inside of the appropriate control structures.

Below are some examples on how to use stream interfaces for some peripherals available on the DE1SoC:

```
val imgIn  = StreamIn[Pixel16](target.VideoCamera) // Input stream for camera
val imgOut = BufferedOut[Pixel16](target.VGA) // Output VGA display
val switch = target.SliderSwitch
val swInput = StreamIn[sw3](switch)
```

More on stream interfaces TBA.

Next, *learn how to build a more complicated Spatial app, Dot Product*.

2.3 1. Vector Inner Product

2.3.1 Catalog of Features

In this section, you will learn about the following components in Spatial:

- Tiling
- Reduce and Fold
- Sequential execution and Coarse-grain pipelining
- Parallelization
- Basic buffering and banking

2.3.2 Application Overview

Inner product (also called dot product) is an extremely simple linear algebra kernel, defined as the sum of the element-wise products between two vectors of data. For this example, we'll assume that the data in this case are 32-bit signed fixed point numbers with 8 fractional bits. You could, however, also do the same operations with custom struct types.

2.3.3 Data Setup and Validation

Let's start by creating the data structures above the Accel that we will compute the dot product on. We will expose the length of these vectors as a command-line argument. We will also write the code below the Accel to ensure we have the correct result:

```
import spatial.dsl._
import org.virtualized._

object DotProduct extends SpatialApp {

  @virtualize
  def main() {

type T = FixPt[TRUE,_24,_8]

val N = args(0).to[Int]
val length = ArgIn[Int]
setArg(length, N)
val result = ArgOut[T]

val vector1_data = Array.tabulate(N){i => random[T](5)}
val vector2_data = Array.tabulate(N){i => random[T](5)}

val vector1 = DRAM[T](length) // DRAMs can be sized by ArgIns
val vector2 = DRAM[T](length)

setMem(vector1, vector1_data)
setMem(vector2, vector2_data)

Accel {}

val result_dot = getArg(result)
val gold_dot = vector1_data.zip(vector2_data){_*_}.reduce{_+_}
val cksum = gold_dot == result_dot
println("Received " + result_dot + ", wanted " + gold_dot)
println("Pass? " + cksum)
  }
```

2.3.4 Tiling and Reduce/Fold

Now we will focus our attention on writing the accelerator code. We must first figure out how to process a variable-sized vector on a fixed hardware design. To do this, we use tiling. Let's create a val for the tileSize just inside the `main()` method:

```
val tileSize = 64
```

Now we can break the vectors into 64-element chunks, and then process these chunks locally on the FPGA using the `Reduce` construct:

```
Accel {
  result := Sequential.Reduce(Reg[T](0))(length by tileSize){tile => // Returns Reg[T], writes␣
↪to ArgOut
    val tile1 = SRAM[T](tileSize)
    val tile2 = SRAM[T](tileSize)

    tile1 load vector1(tile :: tile + tileSize)
    tile2 load vector2(tile :: tile + tileSize)

        val local_accum = Reg[T](0)
    Reduce(local_accum)(tileSize by 1){i => tile1(i) * tile2(i)}{_+_} // Accumulates directly␣
↪into local_accum
        local_accum
```

```
  }{_+_}
}
```

It might seem a bit odd at first that we have the line *{_+_}* twice in this app. This is because the inner reduce accumulates over a tile, and the outer reduce accumulates each result we get from each tile. The `Reduce` construct assumes that the register that is doing the accumulation will effectively reset on each iteration of its parent controller. This means that on the first iteration of the innermost reduce, the hardware will write the first product directly to the register. On the second iteration of this innermost reduce, it will write the current value of local_accum PLUS the next product. This means that if local_accum were declared to have an initial value of 5, reducing on top of it will start at 0, not 5.

Also, notice the `Sequential` modifier on the outer reduce loop. This ensures that everything inside of this loop will happen sequentially and without coarse-grained pipelining. We will play more with this parameter in the next section, *Pipelining and Parallelization*.

The `Reduce` construct takes four pieces of information: **1)** an existing Reg or a new Reg in which to accumulate, **2)** a range and step for its counter to scan, **3)** a map function, and **4)** a reduce function. If a new Reg is declared inside the `Reduce`, then the structure returns this register. Importantly, note that a declaration of `val local_accum = Reg[T](0)` does not mean that local_accum is reset on every iteration. This is a declaration of hardware and is always present. It is the contract implicit with the `Reduce` construct that effectively resets the register. You can manually reset the register in the code with `local_accum.reset`.

Alternatively, you can express the Accel for dot product using a `Fold`. This is similar to a `Reduce`, except the Reg is persistent and not reset unless explicitly reset by the user. In the case where a Reg was declared to have an initial value of 5, the Fold on top of this Reg would start at 5 and not 0. The code would look like this:

```
Accel {
    val accum = Reg[T](0)
    Sequential.Foreach(length by tileSize){tile =>
        val tile1 = SRAM[T](tileSize)
        val tile2 = SRAM[T](tileSize)

        tile1 load vector1(tile :: tile + tileSize)
        tile2 load vector2(tile :: tile + tileSize)

        Fold(accum)(tileSize by 1){i => tile1(i) * tile2(i)}{_+_}
    }
    result := accum
}
```

Let's take a look at the hardware we have generated. The animation below demonstrates how this code will synthesize and execute.

While the above code appears to be correct, there is a problem when handling edge-cases. If the user inputs a vector size that is not a multiple of our tileSize, then we will have an issue with the above code on the final iteration.

To fix this, we need to keep track of how many elements we *actually* want to reduce over each time we execute the inner pipe:

```
Accel {
    val accum = Reg[T](0)
    Sequential.Foreach(length by tileSize){tile =>
        val numel = min(tileSize.to[Int], length - tile)
        val tile1 = SRAM[T](tileSize)
        val tile2 = SRAM[T](tileSize)

        tile1 load vector1(tile :: tile + numel)
        tile2 load vector2(tile :: tile + numel)

        Fold(accum)(numel by 1){i => tile1(i) * tile2(i)}{_+_}
```

```
    }
    result := accum
}
```

2.3.5 Pipelining and Parallelization

Now we will look into ways to speed up the application we have written above.

The first technique is to pipeline the algorithm. In the animation in the previous section, you will notice that the entire hardware is working on one tile at a time. It is possible to pipeline this algorithm at a coarse level such that we overlap the tile loading with the computation. While this boils down to a "prefetching" operation in this particular design, Spatial allows you to arbitrarily pipeline any operations you have in your algorithm and at any level and over any depth.

In order to exploit this technique, you simply need to remove the `Sequential` modifier on the outer loop. By default, all controllers will pipeline their children controllers if no modifiers are added. In this dot product, there are two child stages inside the outer pipe (parallel load of tiles 1 and 2 is the first stage, and reduction over the tiles is the second stage). This kind of coarse-grain pipeline is implemented using asynchronous handshaking signals between each child stage and their respective parent. The resulting code looks like this:

```
Accel {
    val accum = Reg[T](0)
    Foreach(length by tileSize){tile =>
        val numel = min(tileSize.to[Int], length - tile)
        val tile1 = SRAM[T](tileSize)
        val tile2 = SRAM[T](tileSize)

        tile1 load vector1(tile :: tile + numel)
        tile2 load vector2(tile :: tile + numel)

        Fold(accum)(numel by 1){i => tile1(i) * tile2(i)}{_+_}
    }
    result := accum
}
```

This code is expressed in the following animation. Notice that the on-chip SRAM is now larger as it consists of a double buffer. This buffer is what protects one stage of the pipeline from the next. In order to load the next tile into memory, we must retain the data from the previous tile in such a way that the second stage can consume it. While this pipelining improves performance, it consumes more area. Spatial will automatically buffer all SRAMs, Regs, and RegFiles for the user up to whatever depth is required to guarantee correctness. Note that while it is not shown in the animation, the accumulating register is also duplicated, such that one of the duplicates is a double buffer to guarantee correctness for its reader.

We will now look at parallelization as another technique to speed up the algorithm. We will return to the version that uses two `Reduce` nodes rather than the version that uses the `Fold`, and this switch will make sense by the end of the tutorial.

You can think of parallelization of a controller as extending the counter value to hold multiple consecutive values at once. Specifically, if we parallelize the innermost controller, whose counter value is captured by the variable `i`, then this `i` no longer holds a single value. It becomes a vector of consecutive values. If the parallelization is set to 4, then it will hold 4 consecutive values and the controller will complete its execution in a quarter of the time.

Because `i` is used to index into our SRAMs, we need to physically bank our memories in order to ensure that we can read all of the requested values at the same time. The scratchpad memories on-chip have a single write port and a single read port, but the language allows the user to read and write to a memory at will. The Spatial compiler figures out the physical banking, muxing, and duplication of memories that is necessary to ensure the user gets the correct

logical behavior specified in the application. The compiler also generates the necessary reduction tree and parallel hardware required to feed the reduction loop. The animation below demonstrates this innermost parallelization.

Finally, the language also exposes parallelization at controllers beyond the innermost ones. In this particular application, the outer `Reduce` can be parallelized, enabling us to operate on multiple tiles at the same time in parallel. When loops containing other controllers and operations are parallelized, the compiler automatically unrolls the body and duplicates whatever hardware is necessary. It routes the proper lanes of the counter to each of the unrolled bodies and executes them in parallel. Below is an animation depicting this mode of operation.

Notice that the accumulator in stage 2 is now double-buffered. This is because the final reduction stage of the outer reduce is actually viewed as a third stage in the hierarchical control scheme. This means that we need to protect whatever value is in the accumulator when the buffer switches and the third stage prepares to reduce and consume the partial sums.

The reason we could not use the `Fold` version with outer parallelization is because it would require us to have multiple controllers all competing to write to the same register. When there is outer-level parallelization, anything declared inside the body of the controller goes along for the ride when unrolled. This is why we must declare the SRAMs inside of the outer loop. In the case of the `Fold` app, we had to declare the accumulator above the outer loop so that it is visible at the end when we write the result to the ArgOut. Using an outer reduce lets us work on multiple tiles in parallel and merge their results in the final stage of the controller.

2.3.6 Final Code

Finally, below is the complete app that includes all of the performance-oriented features outlined in this page of the tutorial. Refer back to the **'Compiling'_** and **'Synthesizing and Testing'_** (TODO: fix links) sections on the previous page for a refresher on how to test your app.:

```
import spatial.dsl._
import org.virtualized._

object DotProduct extends SpatialApp {

  @virtualize
  def main() {

    type T = FixPt[TRUE,_24,_8]
    val tileSize = 64

    val N = args(0).to[Int]
    val length = ArgIn[Int]
    setArg(length, N)
    val result = ArgOut[T]

    val vector1_data = Array.tabulate(N){i => random[T](5)}
    val vector2_data = Array.tabulate(N){i => random[T](5)}

    val vector1 = DRAM[T](length) // DRAMs can be sized by ArgIns
    val vector2 = DRAM[T](length)

    setMem(vector1, vector1_data)
    setMem(vector2, vector2_data)


    Accel {
        result := Reduce(Reg[T](0))(length by tileSize par 2){tile =>
            val numel = min(tileSize.to[Int], length - tile)
            val tile1 = SRAM[T](tileSize)
            val tile2 = SRAM[T](tileSize)
```

```
            tile1 load vector1(tile :: tile + numel)
            tile2 load vector2(tile :: tile + numel)

            Reduce(Reg[T](0))(numel by 1 par 4){i => tile1(i) * tile2(i)}{_+_}
        }{_+_}
    }

    val result_dot = getArg(result)
    val gold_dot = vector1_data.zip(vector2_data){_*_}.reduce{_+_}
    val cksum = gold_dot == result_dot
    println("Received " + result_dot + ", wanted " + gold_dot)
    println("Pass? " + cksum)
  }
}
```

When you understand the concepts introduced in this page, you may move on to the next example, *3. General Matrix Multiply (GEMM)*, where you will learn to perform reductions on memories, include instrumentation hooks to help balance your pipeline, and see more complicated examples of banking.

2.4 3. General Matrix Multiply (GEMM)

2.4.1 Catalog of Features

In this section, you will learn about the following components in Spatial:

- MemReduce and MemFold
- Instrumentation Hooks
- Advanced Banking
- Advanced Buffering

2.4.2 Application Overview

General Matrix Multiply (GEMM) is a common algorithm in linear algebra, machine learning, statistics, and many other domains. It provides a more interesting trade-off space than the previous tutorial, as there are many ways to break up the computation. This includes using blocking, inner products, outer products, and systolic array techniques. In this tutorial, we will demonstrate how to build a blocked GEMM app that uses outer products, and leave it to the user to try and build a GEMM version that uses inner products. Later tutorials will show how to use shift registers and systolic arrays in other applications, but the same techniques can be retroactively applied to this tutorial on GEMM as well.

2.4.3 Data Setup and Validation

Let's start by creating the data structures above the Accel that we will set up the matrices and compute the gold check. We will expose the dimensions of the matrices as command-line arguments.

```
import spatial.dsl._
import org.virtualized._

object GEMM extends SpatialApp {

  @virtualize
  def main() {
```

```
            type T = FixPt[TRUE,_24,_8]

        val M = ArgIn[Int]
        val N = ArgIn[Int]
        val K = ArgIn[Int]
        setArg(M,args(0).to[Int])
        setArg(N,args(1).to[Int])
        setArg(K,args(2).to[Int])

        val a_data = (0::args(0).to[Int], 0::args(2).to[Int]){(i,j) => random[T](3)}
        val b_data = (0::args(2).to[Int], 0::args(1).to[Int]){(i,j) => random[T](3)}
        val c_init = (0::args(0).to[Int], 0::args(1).to[Int]){(i,j) => 0.to[T]}
        val a = DRAM[T](M, K)
        val b = DRAM[T](K, N)
        val c = DRAM[T](M, N)

        setMem(a, a_data)
        setMem(b, b_data)
        setMem(c, c_init)

            Accel {}

            val accel_matrix = getMatrix(c)

            val gold_matrix = (0::args(0).to[Int], 0::args(1).to[Int]){(i,j) =>
                    Array.tabulate(args(2).to[Int]){k => a_data(i,k) * b_data(k,j)}.reduce{_+_}
            }

            printMatrix(accel_matrix, "Received: ")
            printMatrix(gold_matrix, "Wanted: ")
            val cksum = accel_matrix.zip(gold_matrix){_==_}.reduce{_&&_}
            println("Pass? " + cksum)
  }
}
```

Notice that we create an initial matrix for the result and set all values to 0. This is necessary because GEMM using outer products computes part of a tile of the result and accumulates this on top of what was previously in that tile. This means we will need to fetch a tile from off-chip DRAM and accumulate a new result on top of that, then write this new tile back.

2.4.4 MemReduce and MemFold

The animation below shows how to compute GEMM without tiling, using outer products.

Because we cannot create hardware to handle variable-sized matrices, we must tile the problem. The animation below shows one valid scheme for doing so. We will set our tile sizes in the M, N, and K dimensions above the Accel as follows:

```
val tileM = 16
val tileN = 16
val tileK = 16
```

Note that this is not necessarily the most efficient implementation of this algorithm. It is simply meant to be an implementation that demonstrates features of Spatial.

Now let's write the code to implement this computation. The large arrows and boxes represent matrix multiplies on the highlighted tiles using outer products. There will be six nested loops: one for each dimension of tiling and one for each dimension within the tile.

Considering the tiling loops first, this particular animation shows that we are treating the N dimension as the innermost loop, followed by the M dimension, and finally the K dimension. Below shows the nested loops along with the data structures and their tile transfers required within each scope. Remember that you may add parallelization wherever you please:

```
Accel {
        Foreach(K by tileK){kk =>
                val numel_k = min(tileK.to[Int], K - kk)
                Foreach(M by tileM){mm =>
                        val numel_m = min(tileM.to[Int], M - mm)
                        val tileA_sram = SRAM[T](tileM, tileK)
                        tileA_sram load a(mm::mm+numel_m, kk::kk+numel_k)
                        Foreach(N by tileN){nn =>
                                val numel_n = min(tileN.to[Int], N - nn)
                                val tileB_sram = SRAM[T](tileK, tileN)
                                val tileC_sram = SRAM.buffer[T](tileM, tileN)
                                tileB_sram load b(kk::kk+numel_k, nn::nn+numel_n)
                                tileC_sram load c(mm::mm+numel_m, nn::nn+numel_n)


                                c(mm::mm+numel_m, nn::nn+numel_n) store tileC_sram
                        }
                }
        }
}
```

Note that we must compute the `numel_*` values to handle the edge cases correct, when the tile dimensions do not evenly divide the full matrices.

Also note that we declare `tileC_sram` as a *.buffer* SRAM. If you do not declare it this way, then the compiler will throw an error about this and explain the issue. You will learn more about this in the *Advanced Buffering* section below.

Next, we will implement the full outer product of the tiles that we have brought into the chip:

```
Accel {
        Foreach(K by tileK){kk =>
                val numel_k = min(tileK.to[Int], K - kk)
                Foreach(M by tileM){mm =>
                        val numel_m = min(tileM.to[Int], M - mm)
                        val tileA_sram = SRAM[T](tileM, tileK)
                        tileA_sram load a(mm::mm+numel_m, kk::kk+numel_k)
                        Foreach(N by tileN){nn =>
                                val numel_n = min(tileN.to[Int], N - nn)
                                val tileB_sram = SRAM[T](tileK, tileN)
                                val tileC_sram = SRAM.buffer[T](tileM, tileN)
                                tileB_sram load b(kk::kk+numel_k, nn::nn+numel_n)
                                tileC_sram load c(mm::mm+numel_m, nn::nn+numel_n)

                                MemFold(tileC_sram)(numel_k by 1){k =>
                                        val tileK_local = SRAM[T](tileM, tileN)
                                        Foreach(numel_m by 1, numel_n by 1){(i,j) =>
                                                tileK_local(i,j) = tileA_sram(i,k) * tileB_
↪sram(k,j)
                                        }
                                        tileK_local
                                }{_+_}

                                c(mm::mm+numel_m, nn::nn+numel_n) store tileC_sram
                        }
                }
        }
}
```

Notice that the code added in the above snippet uses a `MemFold` and creates a new memory called `tileK_local`

inside of it. The `MemFold` is similar to the `Fold` used in the previous *1. Vector Inner Product* example, except it operates on SRAMs and RegFiles rather than Regs. The SRAM returned in the body of the map function of the `MemFold` must match the dimensions of the accumulating SRAM given to the controller.

There is also a `MemReduce` node, which is analogous to the `Reduce` node for Regs, but this particular node will not work in this design because we need to accumulate a new partial sum on top of the partial sum that was previously stored for a particular tile in DRAM. The `MemReduce` controller will directly write the result of the map function on the first iteration of the controller (i.e.- when k == 0), and then respect the lambda function (i.e.- addition) for every iteration after that.

2.4.5 Advanced Buffering

This Accel above already implements coarse-grain pipelining at various levels. For example, the controller whose counter is `nn` has three stages in it. The first stage loads `tileB_sram` and tileC_sram`` in parallel, the second stage performs the `MemFold` into `tileC_sram`, and the third stage writes the resulting `tileC_sram` back into the appropriate region of DRAM. This is an example where the compiler will create a triple-buffer for `tileC_sram` in order to ensure that the correct values are being worked with when this coarse-grain pipeline fills up and executes.

If you had not declared `tileC_sram` as a *.buffer* SRAM, then the compiler is suspicious of your code. This is because it is generally very easy when specifying pipelined hardware to accidentally create loop-carry dependency issues. Specifically, in this code, it sees that you write to the SRAM in the first stage, and then write to it again in the second stage. It is very easy, even for advanced users, to write this kind of structure without realizing it and then receive an incorrect result when using a cycle-accurate simulator of the hardware because of values "rotating" through the buffer inadvertently.

The animation below specifically demonstrates the triple buffer `tileC_sram` in this algorithm.

Note that at the beginning and end of each row, there are a few iterations where parts of the buffer are not being used. This is because of the way the loops are written, such that we step through each tile in the N dimension before we increment the tile for M. If you want to write the app such that there are no wasteful fill and drain iterations, you must combine loops appropriately.

2.4.6 Advanced Banking

Let's now add in more optimizations to improve the performance of this application. Specifically, we will parallelize two of the loops in such a way to expose hierarchical banking. The following code shows the loops for `k` and `j` parallelized by 2 and 4 respectively.:

```
Accel {
        Foreach(K by tileK){kk =>
                val numel_k = min(tileK.to[Int], K - kk)
                Foreach(M by tileM){mm =>
                        val numel_m = min(tileM.to[Int], M - mm)
                        val tileA_sram = SRAM[T](tileM, tileK)
                        tileA_sram load a(mm::mm+numel_m, kk::kk+numel_k)
                        Foreach(N by tileN){nn =>
                                val numel_n = min(tileN.to[Int], N - nn)
                                val tileB_sram = SRAM[T](tileK, tileN)
                                val tileC_sram = SRAM.buffer[T](tileM, tileN)
                                tileB_sram load b(kk::kk+numel_k, nn::nn+numel_n)
                                tileC_sram load c(mm::mm+numel_m, nn::nn+numel_n)

                                MemFold(tileC_sram)(numel_k by 1 par 2){k =>
                                        val tileK_local = SRAM[T](tileM, tileN)
                                        Foreach(numel_m by 1, numel_n by 1 par 4){(i,j) =>
                                                tileK_local(i,j) = tileA_sram(i,k) * tileB_
↪sram(k,j)
```

```
                                        }
                                        tileK_local
                                }{_+_}

                                c(mm::mm+numel_m, nn::nn+numel_n) store tileC_sram
                        }
                }
        }
}
```

Now let's look at what happens to `tileB_sram`. It's first and second indices are both parallelized. Index `j` is vectorized by 4, while index `k` is duplicated for two different values of k when the loop is unrolled by 2. This means we must bank `tileB_sram` in both the horizontal and vertical dimensions in order to guarantee that all 8 of these accesses will be able to touch unique banks every time we read from this memory. The animation below demonstrates how we hierarchically bank this SRAM.

Let's consider the situation if we instead decided to parallelize a different way. Below is the code for the application if we chose to parallelize the loading of tileB_sram by 8 while also parallelizing the `k` loop by 2.:

```
Accel {
        Foreach(K by tileK){kk =>
                val numel_k = min(tileK.to[Int], K - kk)
                Foreach(M by tileM){mm =>
                        val numel_m = min(tileM.to[Int], M - mm)
                        val tileA_sram = SRAM[T](tileM, tileK)
                        tileA_sram load a(mm::mm+numel_m, kk::kk+numel_k)
                        Foreach(N by tileN){nn =>
                                val numel_n = min(tileN.to[Int], N - nn)
                                val tileB_sram = SRAM[T](tileK, tileN)
                                val tileC_sram = SRAM.buffer[T](tileM, tileN)
                                tileB_sram load b(kk::kk+numel_k, nn::nn+numel_n par 8)
                                tileC_sram load c(mm::mm+numel_m, nn::nn+numel_n)

                                MemFold(tileC_sram)(numel_k by 1 par 2){k =>
                                        val tileK_local = SRAM[T](tileM, tileN)
                                        Foreach(numel_m by 1, numel_n by 1){(i,j) =>
                                                tileK_local(i,j) = tileA_sram(i,k) * tileB_
↪sram(k,j)
                                        }
                                        tileK_local
                                }{_+_}

                                c(mm::mm+numel_m, nn::nn+numel_n) store tileC_sram
                        }
                }
        }
}
```

While the hierarchical banking scheme shown above will still work for this case, where we have 2 banks along the rows and 8 banks along the columns, the Spatial compiler will perform a memory-saving optimization called Diagonal Banking. In this example, we need to be able to access 8 elements along the column simultaneously, and later in the app we need to access 2 elements from different rows simultaneously. However, these accesses do not occur at the same time, so we do not need 16 unique banks (as is implied by the previous example) and can get away with 8 banks.

If the parallelizations of the various accesses are not multiples of each other, the compiler will figure out the most minimalistic banking scheme that guarantees correctness.

2.4.7 Final Code

Below is the complete GEMM app. See the *HelloWorld* page for a refresher on how to compile and test an app:

```
import spatial.dsl._
import org.virtualized._

object GEMM extends SpatialApp {

  @virtualize
  def main() {

            type T = FixPt[TRUE,_24,_8]
            val tileM = 16
            val tileN = 16
            val tileK = 16

        val M = ArgIn[Int]
        val N = ArgIn[Int]
        val K = ArgIn[Int]
        setArg(M,args(0).to[Int])
        setArg(N,args(1).to[Int])
        setArg(K,args(2).to[Int])

        val a_data = (0::args(0).to[Int], 0::args(2).to[Int]){(i,j) => random[T](3)}
        val b_data = (0::args(2).to[Int], 0::args(1).to[Int]){(i,j) => random[T](3)}
        val c_init = (0::args(0).to[Int], 0::args(1).to[Int]){(i,j) => 0.to[T]}
        val a = DRAM[T](M, K)
        val b = DRAM[T](K, N)
        val c = DRAM[T](M, N)

        setMem(a, a_data)
        setMem(b, b_data)
        setMem(c, c_init)

            Accel {
                    Foreach(K by tileK){kk =>
                            val numel_k = min(tileK.to[Int], K - kk)
                            Foreach(M by tileM){mm =>
                                    val numel_m = min(tileM.to[Int], M - mm)
                                    val tileA_sram = SRAM[T](tileM, tileK)
                                    tileA_sram load a(mm::mm+numel_m, kk::kk+numel_k)
                                    Foreach(N by tileN){nn =>
                                            val numel_n = min(tileN.to[Int], N - nn)
                                            val tileB_sram = SRAM[T](tileK, tileN)
                                            val tileC_sram = SRAM.buffer[T](tileM, tileN)
                                            tileB_sram load b(kk::kk+numel_k, nn::nn+numel_n par␣
↪8)
                                            tileC_sram load c(mm::mm+numel_m, nn::nn+numel_n)

                                            MemFold(tileC_sram)(numel_k by 1 par 2){k =>
                                                    val tileK_local = SRAM[T](tileM, tileN)
                                                    Foreach(numel_m by 1, numel_n by 1){(i,j) =>
                                                            tileK_local(i,j) = tileA_sram(i,k) *␣
↪tileB_sram(k,j)
                                                    }
                                                    tileK_local
                                            }{_+_}

                                            c(mm::mm+numel_m, nn::nn+numel_n) store tileC_sram
                                    }
                            }
                    }
            }

            val accel_matrix = getMatrix(c)
```

```
            val gold_matrix = (0::args(0).to[Int], 0::args(1).to[Int]){(i,j) =>
                    Array.tabulate(args(2).to[Int]){k => a_data(i,k) * b_data(k,j)}.reduce{_+_}
            }

            printMatrix(accel_matrix, "Received: ")
            printMatrix(gold_matrix, "Wanted: ")
            val cksum = accel_matrix.zip(gold_matrix){_==_}.reduce{_&&_}
            println("Pass? " + cksum)

  }
}
```

2.4.8 Instrumentation Hooks

Now that you have finished writing an algorithm, you will want to try to get the best performance possible. In order to get optimal performance, it is important to balance the stages in your pipelines. While you could get a good estimate by eyeballing your code, there is a way to get actual execution cycles on a controller-by-controller basis using a Spatial/special feature called "instrumentation."

To turn on instrumentation hooks, use the `bin/spatial <app name> --synth --instrument` flag when compiling the app. This flag injects performance counters that count the number of cycles each controller is enabled, as well as the number of times a particular controller is done. Note that performance counters will only be injected in the –synth backend.

Once you compile your app, you should run it normally with the run.sh script. You may notice that there are some extra lines that are spitting out information about the app. Running the run.sh script created a file in your current directory called *instrumentation.txt*, which will be used to populate a visualization of your app. Let's start by opening up the controller tree:

```
google-chrome controller_tree.html # Or whatever your favorite browser is (firefox, etc.)
```

You will get a screen that looks like this.

If you play around with this screen, you will see that this shows you the control hierarchy in your app, and points each box back to the original source code. To make this a more useful tool, we will now inject the instrumentation results into this page. Run the script:

```
bash scripts/instrument.sh
```

Now refresh the controller tree page. There should be a lot of red text, similar to the image shown below:

You can now play around with this page and look at how the various stages in your pipelines are performing. We leave it up to the user to figure out how to use parallelizations and rewrite portions of the app to figure out how to balance the pipelines and get better performance.

When you understand the concepts introduced in this page, you may move on to the next example, *3. Differentiator & Sobel Filter*, where you will learn to perform reductions on memories, include instrumentation hooks to help balance your pipeline, and see more complicated examples of banking.

2.5 3. Differentiator & Sobel Filter

2.5.1 Catalog of Features

In this section, you will learn about the following components in Spatial:

- LineBuffer

- ShiftRegister
- LUT
- Spatial Functions and Multifile Projects

2.5.2 Application Overview

Convolution is a common algorithm in linear algebra, machine learning, statistics, and many other domains. The tutorials in this section will demonstrate how to use the building blocks that Spatial provides to do convolutions.

Specifically, we will build a basic differentiator for a time-series using sliding window averaging for an example 1D convolution. The animation below demonstrates this convolution (credit http://pages.jh.edu/~signals/convolve/index.html).

We will then build a Sobel Filter to detect edges on an image for an example of 2D convolution. A Sobel filter is the average of the convolution with the following two kernels:

```
KernelV:
  1   2   1
  0   0   0
 -1  -2  -1

KernelH:
  1   0  -1
  2   0  -2
  1   0  -1
```

While this section will not explore convolutions in more than 2 dimensions, it is possible to combine the Spatial primitives demonstrated in this section and previous sections to build up a higher-dimensional convolution. The animation below demonstrates the 2D convolution with the padding that we will use (credit https://github.com/vdumoulin/conv_arithmetic). Alternatively, Spatial supports 2D convolutions as matrix multiplies. See (TODO: Link to "toeplitz" API) for more details.

TODO: Fix right and bottom padding to match app below

2.5.3 Data Setup and Validation

Let's start by creating the data structures above the Accel that we will set up the image and filters and compute the gold check. We will expose the rows of the images as command-line arguments.:

```
import spatial.dsl._
import org.virtualized._

object DiffAndSobel extends SpatialApp {

  @virtualize
  def main() {

    val R = args(0).to[Int]
    val C = args(1).to[Int]
    val vec_len = args(2).to[Int]
    val vec_tile = 64
    val maxcols = 128 // Required for LineBuffer
    val ROWS = ArgIn[Int]
    val COLS = ArgIn[Int]
    val LEN = ArgIn[Int]
```

```
    setArg(ROWS,R)
    setArg(COLS,C)
    setArg(LEN, vec_len)

    val window = 16
    val x_t = Array.tabulate(vec_len){i =>
      val x = i.to[T] * (4.to[T] / vec_len.to[T]).to[T] - 2
      -0.18.to[T] * pow(x, 4) + 0.5.to[T] * pow(x, 2) + 0.8.to[T]
    }
    val h_t = Array.tabulate(16){i => if (i < window/2) 1.to[T] else -1.to[T]}
    printArray(x_t, "x_t data:")

    val X_1D = DRAM[T](LEN)
    val H_1D = DRAM[T](window)
    val Y_1D = DRAM[T](LEN)
    setMem(X_1D, x_t)
    setMem(H_1D, h_t)

    val border = 3
    val image = (0::R, 0::C){(i,j) => if (j > border && j < C-border && i > border && i < C -␣
↪border) (i*16).to[T] else 0.to[T]}
    printMatrix(image, "Image:")
    val kernelv = Array[T](1,2,1,0,0,0,-1,-2,-1)
    val kernelh = Array[T](1,0,-1,2,0,-2,1,0,-1)
    val X_2D = DRAM[T](ROWS, COLS)
    val Y_2D = DRAM[T](ROWS, COLS)
    setMem(X_2D, image)

          Accel{}

    val Y_1D_result = getMem(Y_1D)
    val Y_2D_result = getMatrix(Y_2D)

    val Y_1D_gold = Array.tabulate(vec_len){i =>
      Array.tabulate(window){j =>
        val data = if (i - j < 0) 0 else x_t(i-j)
        data * h_t(j)
      }.reduce{_+_}
    }
    val Y_2D_gold = (0::R, 0::C){(i,j) =>
      val h = Array.tabulate(3){ii => Array.tabulate(3){jj =>
        val img = if (i-ii < 0 || j-jj < 0) 0 else image(i-ii,j-jj)
        img * kernelh((2-ii)*3+(2-jj))
      }}.flatten.reduce{_+_}
      val v = Array.tabulate(3){ii => Array.tabulate(3){jj =>
        val img = if (i-ii < 0 || j-jj < 0) 0 else image(i-ii,j-jj)
        img * kernelv((2-ii)*3+(2-jj))
      }}.flatten.reduce{_+_}
      abs(v) + abs(h)
    }

    printArray(Y_1D_result, "1D Result:")
    printArray(Y_1D_gold, "1D Gold:")
    printMatrix(Y_2D_result, "2D Result:")
    printMatrix(Y_2D_gold, "2D Gold:")

    val margin = 0.25.to[T]
    val cksum_1D = Y_1D_result.zip(Y_1D_gold){(a,b) => abs(a - b) < margin}.reduce{_&&_}
    val cksum_2D = Y_2D_result.zip(Y_2D_gold){(a,b) => abs(a - b) < margin}.reduce{_&&_}
    println("1D Pass? " + cksum_1D + ", 2D Pass? " + cksum_2D)
      }
    }
```

Note that there is a val called "maxcols." In the *2D Convolution* section, we will demonstrate how the line buffer works and it will become clear why we must constrain the maximum number of columns in our image for the app to work.

2.5.4 1D Convolution

In order to perform the 1D convolution, we need a pipeline to perform two operations. The first is to load one tile at a time, and the second is to shift data through a window and perform a dot product between this window and the filter. We must also do a one-time load of the filter kernel. The snippet below shows this code:

```
Accel{
  val filter_data = RegFile[T](window)
  filter_data load H_1D
  Foreach(LEN by vec_tile){i =>
    val numel = min(vec_tile.to[Int], LEN-i)
    val x_tile = SRAM[T](vec_tile)
    val y_tile = SRAM[T](vec_tile)
    x_tile load X_1D(i::i+numel)

    val sr1D = RegFile[T](1,window)
    Foreach(numel by 1){j =>
      sr1D(0,*) <<= x_tile(j) // Shift new point into sr1D
      y_tile(j) = Reduce(Reg[T])(window by 1){k =>
        val data = mux(i + j - k < 0, 0.to[T], sr1D(0,k)) // Handle edge case
        data * filter_data(k)
      }{_+_}
    }

    Y_1D(i::i+numel) store y_tile
  }
}
```

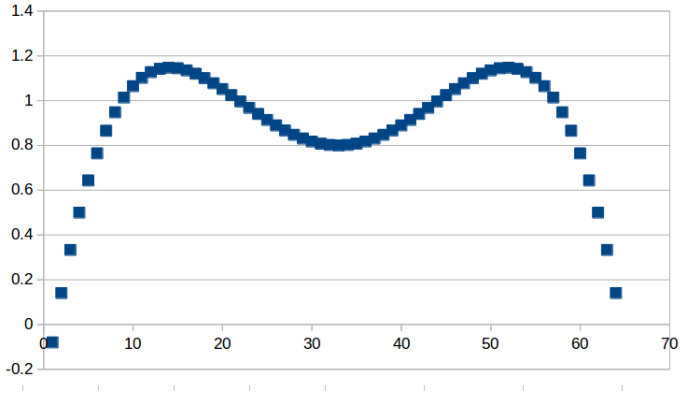
The app above uses familiar concepts described in previous parts of this tutorial, except for the RegFiles. The first RegFile, `filter_data`, is created to hold the filter data. It is equally valid to use an SRAM for this structure, but it is generally more efficient for small memories to use RegFiles, as this reduces the number of wasted addresses in a physical BRAM on-chip. The second RegFile, `sr1D`, is used as a shift register. We use the <<= operator to indicate that we want to shift into it from the entry address (i.e.- address 0), and move all the existing data backwards by one address. Later, we will see how to specify strides for shift registers, as well as shift into an entry plane of a multidimensional shift register.

While this app uses tiles to perform convolution, it is possible to use the shift register in the same way to do convolution on streaming data by directly enqueueing to the shift register. Also, it may not seem completely intuitive that we use the shift register at all, since we can just index into the x_tile directly. However, if you want to parallelize the reduction, the shift register comes fully banked since it is composed of registers. Parallel accesses to the SRAM directly, with a sliding window, will result in lots of SRAM duplication and inefficiency.

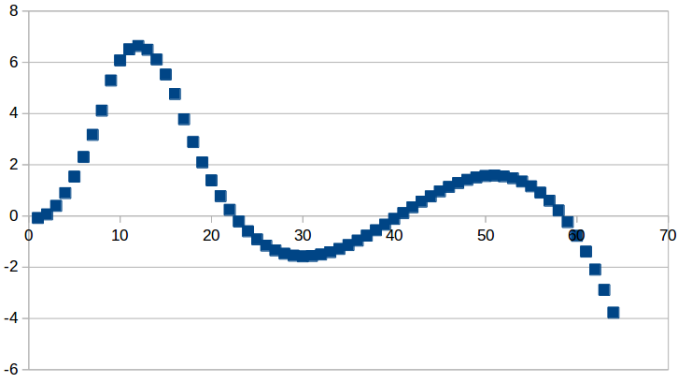
Finally, there is a mux inside the `Reduce` map function. This mux is to check if the data in a particular address of the shift register corresponds to data with a "negative" address in the X_1D data structure.

If you plot the resulting data in a spreadsheet, you should get something that looks like this. We can use these plots as a sanity check for our differentiator kernel.

Input:



Derivative:



2.5.5 2D Convolution

Now we will focus on the Sobel filter that will perform a 2D convolution. First, we will introduce a LineBuffer memory structure. A LineBuffer is a special case on an N-buffered 1D SRAM exposed to the user. It allows one or more rows of DRAM to be buffered into on-chip memory while previous rows can be accessed in a logically-rotating way. A LineBuffer is generally coupled with a shift register, and the animation below shows the specific usage of this pair in this tutorial.

Note that in the last frame, the "buffer" row of the line buffer contains row 7 of the image. This is because this line buffer is physically implemented with four SRAMs and uses access redirection to create the logical behavior shown in the animation. After the last row is loaded and we drain the last frame, the buffers inside the line buffer will rotate but no new line will fill the buffer SRAM, leaving behind the data from row 7 even though it will not get used in this particular case. The Spatial compiler will also determine how to bank and duplicate the SRAMs that compose the line buffer automatically, should you choose to have a strided convolution.

It is also possible now to see why we must set a hard cap on the number of columns in the image if we are to use the line buffer - shift register combination. The logic that handles the rotation of the line buffer rows is tied to the controller hierarchy that manages the writes and reads about the line buffer. If we were to try to tile this operation along the columns, then our line buffer would load one tile of the row into the buffer, while row 0 of the line buffer would contain the previous part of that row. This splitting of a single line is semantically incorrect for convolution.

For this 2D convolution, we also introduce the lookup table (LUT). This is a read-only memory whose values are known at compile time. It is implemented using registers and muxes to index into it.

The snippet below shows how to generate an accel that performs the operations shown above:

```
  Accel {
val lb = LineBuffer[T](3, maxcols)
val sr = RegFile[T](3, 3)
val kernelH = LUT[T](3,3)(1.to[T], 2.to[T], 1.to[T],
                          0.to[T], 0.to[T], 0.to[T],
                         -1.to[T],-2.to[T],-1.to[T])
val kernelV = LUT[T](3,3)(1.to[T], 0.to[T], -1.to[T],
                          2.to[T], 0.to[T], -2.to[T],
                          1.to[T], 0.to[T], -1.to[T])
val lineout = SRAM[T](maxcols)
Foreach(ROWS by 1){row =>
  lb load X_2D(row, 0::COLS)
  Foreach(COLS by 1){j =>
    Foreach(3 by 1 par 3){i => sr(i,*) <<= lb(i,j)}
    val accumH = Reduce(Reg[T](0.to[T]))(3 by 1, 3 by 1){(ii,jj) =>
      val img = if (row - 2 + ii.to[Int] < 0 || j.to[Int] - 2 + jj.to[Int] < 0) 0.to[T] else␣
↪sr(ii, 2 - jj)
      img * kernelH(ii,jj)
    }{_+_}
    val accumV = Reduce(Reg[T](0.to[T]))(3 by 1, 3 by 1){(ii,jj) =>
      val img = if (row - 2 + ii.to[Int] < 0 || j.to[Int] - 2 + jj.to[Int] < 0) 0.to[T] else␣
↪sr(ii, 2 - jj)
      img * kernelV(ii,jj)
    }{_+_}
    lineout(j) = abs(accumV.value) + abs(accumH.value)
  }
  Y_2D(row, 0::COLS) store lineout
}
  }
```

It is possible to improve the performance of this algorithm using parallelization. However, we leave this as an exercise to the user or direct the user to some example apps written in the spatial-apps repository. While parallelizing every loop will speed up this algorithm, some loops will give incorrect results if parallelized while others will maintain the correct result if extra code is added to handle the edge cases appropriately

2.5.6 Spatial Functions and Multifile

Sometimes complicated apps can get very cluttered inside the Accel block so you will want to break your app into multiple functions, possibly across multiple files. Now we will aim to create the following Accel block, where the method calls are defined in a separate file:

```
Accel{
  Conv1D(Y_1D, X_1D, H_1D, window, vec_tile) // Output DRAM, Input Data, Kernel
  Sobel2D(Y_2D, X_2D, maxcols)               // Output DRAM, Input Image
}
```

We can write the functions used above as follows:

```
@virtualize
def Conv1D[T:Type:Num](output: DRAM1[T],
                       input: DRAM1[T],
                       filter: DRAM1[T],
                       window: scala.Int, vec_tile: scala.Int): Unit = {

    val filter_data = RegFile[T](window)
    filter_data load filter
    Foreach(input.size by vec_tile){i =>
      val numel = min(vec_tile.to[Int], input.size-i)
      val x_tile = SRAM[T](vec_tile)
      val y_tile = SRAM[T](vec_tile)
      x_tile load input(i::i+numel)

      val sr1D = RegFile[T](1,window)
```

```
      Foreach(numel by 1){j =>
        sr1D(0,*) <<= x_tile(j) // Shift new point into sr1D
        y_tile(j) = Reduce(Reg[T])(window by 1){k =>
          val data = mux(i + j - k < 0, 0.to[T], sr1D(0,k)) // Handle edge case
          data * filter_data(k)
        }{_+_}
      }

      output(i::i+numel) store y_tile
    }
}


@virtualize
def Sobel2D[T:Type:Num](output: DRAM2[T],
                        input: DRAM2[T], maxcols: scala.Int): Unit = {

    val lb = LineBuffer[T](3, maxcols)
    val sr = RegFile[T](3, 3)
    val kernelH = LUT[T](3,3)(1.to[T], 2.to[T], 1.to[T],
                              0.to[T], 0.to[T], 0.to[T],
                             -1.to[T],-2.to[T],-1.to[T])
    val kernelV = LUT[T](3,3)(1.to[T], 0.to[T], -1.to[T],
                              2.to[T], 0.to[T], -2.to[T],
                              1.to[T], 0.to[T], -1.to[T])
    val lineout = SRAM[T](maxcols)
    Foreach(input.rows by 1){row =>
      lb load input(row, 0::input.cols)
      Foreach(input.cols by 1){j =>
        Foreach(3 by 1 par 3){i => sr(i,*) <<= lb(i,j)}
        val accumH = Reduce(Reg[T](0.to[T]))(3 by 1, 3 by 1){(ii,jj) =>
          val img = if (row - 2 + ii.to[Int] < 0 || j.to[Int] - 2 + jj.to[Int] < 0) 0.to[T] else␣
↪sr(ii, 2 - jj)
          img * kernelH(ii,jj)
        }{_+_}
        val accumV = Reduce(Reg[T](0.to[T]))(3 by 1, 3 by 1){(ii,jj) =>
          val img = if (row - 2 + ii.to[Int] < 0 || j.to[Int] - 2 + jj.to[Int] < 0) 0.to[T] else␣
↪sr(ii, 2 - jj)
          img * kernelV(ii,jj)
        }{_+_}
        lineout(j) = abs(accumV.value) + abs(accumH.value)
      }
      output(row, 0::input.cols) store lineout
    }
}
```

Notice that instead of using the input arguments, `ROWS`, `COLS`, and `LEN`, we can use properties defined on the DRAMs directly.

You can place these functions anywhere inside of your DiffAndSobel object. If you want to place them inside of a separate file entirely, then you simply need to make the *trait* that contains the method definitions extend SpatialApp, and then have the next file create an *object* that extends the first trait:

```
import org.virtualized._
import spatial.dsl._

object AccelFile extends FunctionsFile {

  @virtualize
  def main() {
    Accel {
       FunctionsFile.fcn_call()
    }
}

-------------------------------
```

```
import org.virtualized._
import spatial.dsl._

trait FunctionsFile extends SpatialApp{

  @virtualize
  def fcn_call() {/* do things */}

}
```

2.5.7 Final Code

Below is the final code for a single-file, functionized version of the two convolutions discussed in this tutorial. See the *HelloWorld* page for a refresher on how to compile and test.:

```
import spatial.dsl._
import org.virtualized._

object DiffAndSobel extends SpatialApp {

    @virtualize
    def Conv1D[T:Type:Num](output: DRAM1[T],
                        input: DRAM1[T],
                        filter: DRAM1[T],
                        window: scala.Int, vec_tile: scala.Int): Unit = {

        val filter_data = RegFile[T](window)
        filter_data load filter
        Foreach(input.size by vec_tile){i =>
          val numel = min(vec_tile.to[Int], input.size-i)
          val x_tile = SRAM[T](vec_tile)
          val y_tile = SRAM[T](vec_tile)
          x_tile load input(i::i+numel)

          val sr1D = RegFile[T](1,window)
          Foreach(numel by 1){j =>
            sr1D(0,*) <<= x_tile(j) // Shift new point into sr1D
            y_tile(j) = Reduce(Reg[T])(window by 1){k =>
              val data = mux(i + j - k < 0, 0.to[T], sr1D(0,k)) // Handle edge case
              data * filter_data(k)
            }{_+_}
          }

          output(i::i+numel) store y_tile
        }
    }


    @virtualize
    def Sobel2D[T:Type:Num](output: DRAM2[T],
                            input: DRAM2[T], maxcols: scala.Int): Unit = {

        val lb = LineBuffer[T](3, maxcols)
        val sr = RegFile[T](3, 3)
        val kernelH = LUT[T](3,3)(1.to[T], 2.to[T], 1.to[T],
                                  0.to[T], 0.to[T], 0.to[T],
                                 -1.to[T],-2.to[T],-1.to[T])
        val kernelV = LUT[T](3,3)(1.to[T], 0.to[T], -1.to[T],
                                  2.to[T], 0.to[T], -2.to[T],
                                  1.to[T], 0.to[T], -1.to[T])
        val lineout = SRAM[T](maxcols)
        Foreach(input.rows by 1){row =>
          lb load input(row, 0::input.cols)
```

```
        Foreach(input.cols by 1){j =>
          Foreach(3 by 1 par 3){i => sr(i,*) <<= lb(i,j)}
          val accumH = Reduce(Reg[T](0.to[T]))(3 by 1, 3 by 1){(ii,jj) =>
            val img = if (row - 2 + ii.to[Int] < 0 || j.to[Int] - 2 + jj.to[Int] < 0) 0.to[T]␣
↪else sr(ii, 2 - jj)
            img * kernelH(ii,jj)
          }{_+_}
          val accumV = Reduce(Reg[T](0.to[T]))(3 by 1, 3 by 1){(ii,jj) =>
            val img = if (row - 2 + ii.to[Int] < 0 || j.to[Int] - 2 + jj.to[Int] < 0) 0.to[T]␣
↪else sr(ii, 2 - jj)
            img * kernelV(ii,jj)
          }{_+_}
          lineout(j) = abs(accumV.value) + abs(accumH.value)
        }
        output(row, 0::input.cols) store lineout
      }
    }


    @virtualize
    def main() {

      type T = FixPt[TRUE,_16,_16]

      val R = args(0).to[Int]
      val C = args(1).to[Int]
      val vec_len = args(2).to[Int]
      val vec_tile = 64
      val maxcols = 128 // Required for LineBuffer
      val ROWS = ArgIn[Int]
      val COLS = ArgIn[Int]
      val LEN = ArgIn[Int]
      setArg(ROWS,R)
      setArg(COLS,C)
      setArg(LEN, vec_len)

      val window = 16
      val x_t = Array.tabulate(vec_len){i =>
        val x = i.to[T] * (4.to[T] / vec_len.to[T]).to[T] - 2
        println(" x " + x)
        -0.18.to[T] * pow(x, 4) + 0.5.to[T] * pow(x, 2) + 0.8.to[T]
      }
      val h_t = Array.tabulate(16){i => if (i < window/2) 1.to[T] else -1.to[T]}
      printArray(x_t, "x_t data:")

      val X_1D = DRAM[T](LEN)
      val H_1D = DRAM[T](window)
      val Y_1D = DRAM[T](LEN)
      setMem(X_1D, x_t)
      setMem(H_1D, h_t)

      val border = 3
      val image = (0::R, 0::C){(i,j) => if (j > border && j < C-border && i > border && i < C -␣
↪border) (i*16).to[T] else 0.to[T]}
      printMatrix(image, "image: ")
      val kernelv = Array[T](1,2,1,0,0,0,-1,-2,-1)
      val kernelh = Array[T](1,0,-1,2,0,-2,1,0,-1)
      val X_2D = DRAM[T](ROWS, COLS)
      val Y_2D = DRAM[T](ROWS, COLS)
      setMem(X_2D, image)

      Accel{
        Conv1D(Y_1D, X_1D, H_1D)
        Sobel2D(Y_2D, X_2D)
      }

      val Y_1D_result = getMem(Y_1D)
```

```
    val Y_2D_result = getMatrix(Y_2D)

    val Y_1D_gold = Array.tabulate(vec_len){i =>
      Array.tabulate(window){j =>
        val data = if (i - j < 0) 0 else x_t(i-j)
        data * h_t(j)
      }.reduce{_+_}
    }
    val Y_2D_gold = (0::R, 0::C){(i,j) =>
      val h = Array.tabulate(3){ii => Array.tabulate(3){jj =>
        val img = if (i-ii < 0 || j-jj < 0) 0 else image(i-ii,j-jj)
        img * kernelh((2-ii)*3+(2-jj))
      }}.flatten.reduce{_+_}
      val v = Array.tabulate(3){ii => Array.tabulate(3){jj =>
        val img = if (i-ii < 0 || j-jj < 0) 0 else image(i-ii,j-jj)
        img * kernelv((2-ii)*3+(2-jj))
      }}.flatten.reduce{_+_}
      abs(v) + abs(h)
    }

    printArray(Y_1D_result, "1D Result:")
    printArray(Y_1D_gold, "1D Gold:")
    printMatrix(Y_2D_result, "2D Result:")
    printMatrix(Y_2D_gold, "2D Gold:")

    val margin = 0.25.to[T]
    val cksum_1D = Y_1D_result.zip(Y_1D_gold){(a,b) => abs(a - b) < margin}.reduce{_&&_}
    val cksum_2D = Y_2D_result.zip(Y_2D_gold){(a,b) => abs(a - b) < margin}.reduce{_&&_}
    println("1D Pass? " + cksum_1D + ", 2D Pass? " + cksum_2D)
  }
}
```

2.6 4. Needleman-Wunsch (NW)

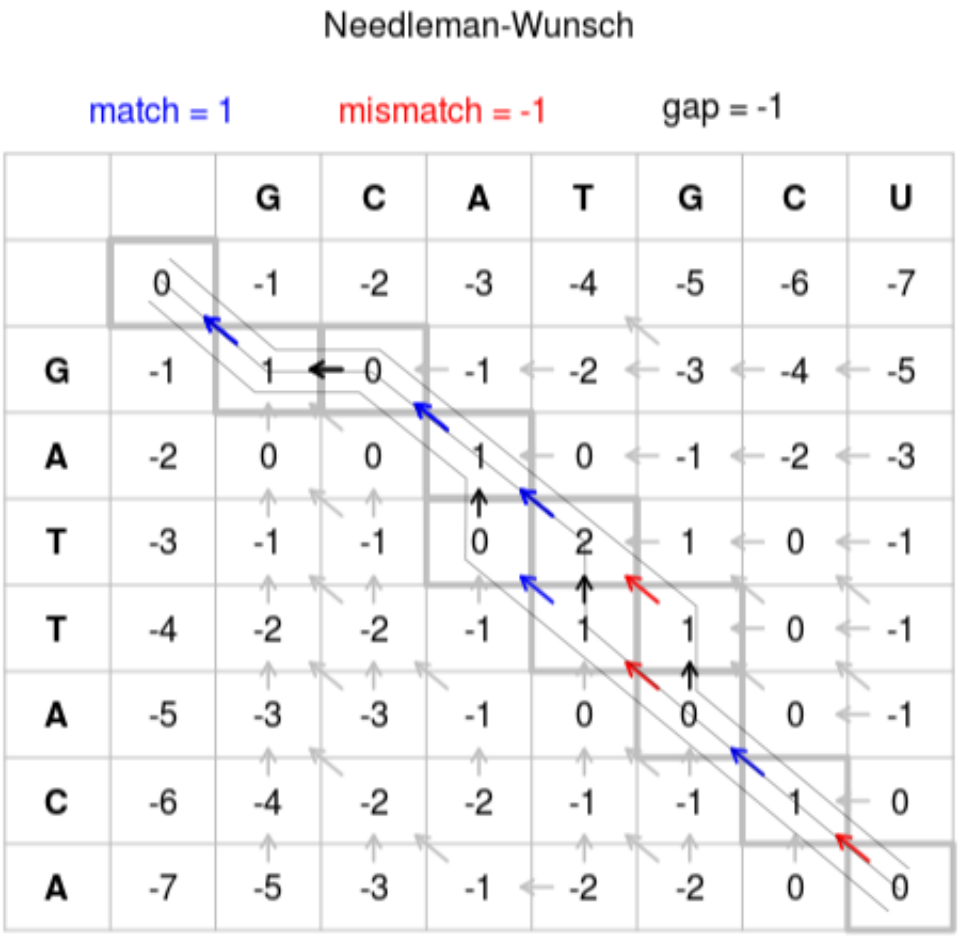
2.6.1 Catalog of Features

In this section, you will learn about the following components in Spatial:

- FSM
- Branching
- FIFO
- Systolic Arrays
- File IO and text management
- Asserts, Breakpoints, and Sleep

2.6.2 Application Overview

The Needleman-Wunsch algorithm is an algorithm used in bioinformatics to align protein or nucleotide sequences. It builds a scoring matrix based on two strings, and then backtraces through the score matrix to determine the alignment of minimum error. For more information on the algorithm's history and implementations, visit the Wikipedia page (https://en.wikipedia.org/wiki/Needleman-Wunsch_algorithm). The image below (credit Wikipedia) demonstrates a rough overview of how the algorithm works.



2.6.3 Data Setup and Validation

In this algorithm, we will assume that a domain-expert keeps data files of DNA sequences, `seqA.txt` and `seqB.txt`. We will create an app that will read whatever text is in these files, pass it to the FPGA, return with the best alignment between the two, and print out the result back into files called `alignedA.txt` and `alignedB.txt`. In order to test if our alignments are "correct," we will aim to have less than 10% of the entries be in error.:

```
import spatial.dsl._
import org.virtualized._

object NW extends SpatialApp {

  @virtualize
  def main() {
    val seqA_text = loadCSV1D[MString]("/home/mattfel/seqA.txt", " ").apply(0) // Loads into␣
↪array with 1 string
    val seqB_text = loadCSV1D[MString]("/home/mattfel/seqB.txt", " ").apply(0) // Loads into␣
↪array with 1 string
    println("Aligning " + seqA_text + " and ")
    println("         " + seqB_text)

    val seqA_data = argon.lang.String.string2num(seqA_text) // Returns array of Int8
    val seqB_data = argon.lang.String.string2num(seqB_text) // Returns array of Int8
```

```
    val seq_length = seqA_data.length
    val LEN = ArgIn[Int]
    val LEN2x = ArgIn[Int]
    setArg(LEN, seq_length)
    setArg(LEN2x, seq_length*2)
    val seqA = DRAM[Int8](LEN)
    val seqB = DRAM[Int8](LEN)
    setMem(seqA, seqA_data)
    setMem(seqB, seqB_data)
    val seqA_aligned = DRAM[Int8](LEN)
    val seqB_aligned = DRAM[Int8](LEN2x)

    val max_length = 256
    val d = argon.lang.String.char2num("-")
    val dash = ArgIn[Int8]
    setArg(dash,d)
    val u = argon.lang.String.char2num("_")
    val underscore = ArgIn[Int8]
    setArg(underscore,u)

    Accel{}

    val seqA_aligned_result = getMem(seqA_aligned)
    val seqB_aligned_result = getMem(seqB_aligned)

    val errors = seqA_aligned_result.zip(seqB_aligned_result){(a,b) => if (a != b && a != d && b␣
↪!= d) 1 else 0}.reduce{_+_}
    println("Found " + errors + " errors out of " + {seq_length*2} + " characters")
    val cksum = errors.to[Float] / (seq_length*2).to[Float] < 0.1.to[Float]
    println("Acceptable? " + cksum)

    val seqA_aligned_string = argon.lang.String.num2string(seqA_aligned_result)
    val seqB_aligned_string = argon.lang.String.num2string(seqB_aligned_result)
    println("Aligned A: " + seqA_aligned_string)
    println("Aligned B: " + seqB_aligned_string)
  }
}
```

2.6.4 Score Matrix Population

In this first section, we will make a forward pass to fill out the score matrix. In this algorithm, we need to embed two pieces of information in each matrix entry: the score at that entry and the direction to travel to achieve that score (N, W, or NW). We will start by defining a new struct that can contain this tuple up above our `main()`:

```
@struct case class nw_tuple(score: Int16, ptr: Int16)
```

As we traverse the score matrix, we check the left, top, and top-left entries, add a score update, and check which path gives us the maximum score for that entry. To determine the additional score when coming from the top-left, we check if the letter at the top of the column (from string A) matches the letter from the left of the row (from string B). If there is a match, then this path is rewarded with an addition of 1. If they do not match, then we penalize this path with a score of -1. We then look at the cost when coming from the left and from the top. These transitions correspond to skipping an entry in B and skipping an entry in A, respectively, and we penalize them as they do not correspond to string matches. This transition is called a "gap." Let's now assign vals to keep track of these properties:

```
val SKIPB = 0 // move left
val SKIPA = 1 // move up
val ALIGN = 2 // move diagonal
val MATCH_SCORE = 1
val MISMATCH_SCORE = -1
val GAP_SCORE = -1
```

Now, we can write the code that will traverse the matrix from top-left to bottom-right and update each entry of the score matrix. Note that along the left edge and the top edge of the score matrix, we initialize the scores by -1 each for each hop away from the top left corner. Then, for each entry, we first compute if there is a match between the elements in string A and string B. We then proceed to compute the `from_left`, `from_top`, and `from_diag` updates based on these values and choose the smallest of them. When getting this result, we keep the tuple that consists of both the new score and the path taken to achieve this new score. Finally, we update the score matrix so that this new value is available for the next update:

```
Accel{
    val seqa_sram_raw = SRAM[Int8](max_length)
    val seqb_sram_raw = SRAM[Int8](max_length)

    seqa_sram_raw load seqA(0::LEN)
    seqb_sram_raw load seqB(0::LEN)

    val score_matrix = SRAM[nw_tuple](max_length+1,max_length+1)

    Foreach(LEN+1 by 1){ r =>
      Sequential.Foreach(0 until LEN+1 by 1) { c =>
        val previous_result = Reg[nw_tuple]
        val update = if (r == 0) (nw_tuple(-c.as[Int16], 0)) else if (c == 0) (nw_tuple(-r.
↪as[Int16], 1)) else {
          val match_score = mux(seqa_sram_raw(c-1) == seqb_sram_raw(r-1), MATCH_SCORE.to[Int16],␣
↪MISMATCH_SCORE.to[Int16])
          val from_top = score_matrix(r-1, c).score + GAP_SCORE
          val from_left = previous_result.score + GAP_SCORE
          val from_diag = score_matrix(r-1, c-1).score + match_score
          mux(from_left >= from_top && from_left >= from_diag, nw_tuple(from_left, SKIPB),␣
↪mux(from_top >= from_diag, nw_tuple(from_top,SKIPA), nw_tuple(from_diag, ALIGN)))
        }
        previous_result := update
        if (c >= 0) {score_matrix(r,c) = update}
      }
    }
}
```

While it is possible to parallelize the row updates in this algorithm, it is a little tricky because you should not update any entry until you have all of its three adjacent source entries. See (TODO: link to spatial-apps) for an example on how to safely parallelize across rows.

2.6.5 Score Matrix Traceback

Now we can traverse the score matrix, starting from the bottom right. We will use a FIFO to store the aligned result, and a finite state machine (FSM) to handle the back trace and complete when the FIFOs are filled. The state in the FSM starts at 0, which we use for the state to trace back through the matrix. When we either hit the top edge or the left edge of the score matrix, we jump to state 1 which is used to pad both of the FIFOs until they fill up. Once the FSM detects that they are full, it exits and the results are stored to DRAM. The branch conditions in this FSM demonstrate how we can use if/then/else to arbitrarily execute parts of the hardware.:

```
val traverseState = 0
val padBothState = 1
val doneState = 2

val seqa_fifo_aligned = FIFO[Int8](max_length*2)
val seqb_fifo_aligned = FIFO[Int8](max_length*2)
val b_addr = Reg[Int](0)
val a_addr = Reg[Int](0)
b_addr := LEN
a_addr := LEN
val done_backtrack = Reg[Bit](false)
FSM[Int](state => state != 2) { state =>
  if (state == traverseState) {
```

```
    if (score_matrix(b_addr,a_addr).ptr == ALIGN.to[Int16]) {
      seqa_fifo_aligned.enq(seqa_sram_raw(a_addr-1), !done_backtrack)
      seqb_fifo_aligned.enq(seqb_sram_raw(b_addr-1), !done_backtrack)
      done_backtrack := b_addr == 1.to[Int] || a_addr == 1.to[Int]
      b_addr :-= 1
      a_addr :-= 1
    } else if (score_matrix(b_addr,a_addr).ptr == SKIPA.to[Int16]) {
      seqb_fifo_aligned.enq(seqb_sram_raw(b_addr-1), !done_backtrack)
      seqa_fifo_aligned.enq(dash, !done_backtrack)
      done_backtrack := b_addr == 1.to[Int]
      b_addr :-= 1
    } else {
      seqa_fifo_aligned.enq(seqa_sram_raw(a_addr-1), !done_backtrack)
      seqb_fifo_aligned.enq(dash, !done_backtrack)
      done_backtrack := a_addr == 1.to[Int]
      a_addr :-= 1
    }
  } else if (state == padBothState) {
    seqa_fifo_aligned.enq(underscore, !seqa_fifo_aligned.full)
    seqb_fifo_aligned.enq(underscore, !seqb_fifo_aligned.full)
  } else {}
} { state =>
  mux(state == traverseState && ((b_addr == 0.to[Int]) || (a_addr == 0.to[Int])), padBothState,
    mux(seqa_fifo_aligned.full || seqb_fifo_aligned.full, doneState, state))
}

seqA_aligned(0::LEN2x) store seqa_fifo_aligned
seqB_aligned(0::LEN2x) store seqb_fifo_aligned
```

Generally, an FSM is a hardware version of a while loop. It allows you to arbitrarily branch between control structures and selectively execute code until some breaking state condition is reached.

2.6.6 Asserts, Breakpoints, and Sleep

There are sometimes cases where the app writer wants to escape the app early or pause the app for a period of time. In this subsection we will explore how to implement the breakpoint/exit and sleep functions in Spatial.

Firstly, we will discuss breakpoints. These could be for debugging purposes, such as determining why a non-deterministic app is hanging on the FPGA, or for practical purposes, such as handling errors when decompressing a faulty JPEG header. Spatial allows the user to insert breakpoints arbitrarily in the code and will exit the application early and report which breakpoint triggered the exit, if any, at runtime.

In this example, we will demonstrate how to use breakpoints in Spatial by assuming the app writer wants to halt the NW algorithm the first time a character in either string A, string B, or neither is skipped and wants to know which of these conditions caused the exit:

```
if (score_matrix(b_addr,a_addr).ptr == ALIGN.to[Int16]) {
  ...
  breakpoint()
} else if (score_matrix(b_addr,a_addr).ptr == SKIPA.to[Int16]) {
  ...
  assert(score_matrix(b_addr,a_addr).ptr != SKIPA.to[Int16], "This is an assert example")
} else {
  ...
  exit()
}
```

Note that "breakpoint()" in this case is not the same as a breakpoint in software. A breakpoint here causes the entire app to quit, rather than allowing the user to step through code manually. While functionality to switch from the FPGA's built in clock to a manual clock to let the user manually step through cycles may be implemented in the future, there are no current plans to support this.

The above code may generate output that looks like this if the second breakpoint were reached first (breakpoints are 0-indexed):

```
==================
  Breakpoint 1 triggered!
    tutorial.scala:100:23 - This is an assert example
==================
```

In apps that interact with real external systems, such as pixel buffers, audio devices, and sensors, it may be very useful to make the FPGA stall for a period of time so that it interacts properly with these systems. It can also be useful in debugging, to slow down the speed at which a piece of code executes. While grad students may not get much sleep, Spatial makes it easy to put your FPGA to sleep:

```
sleep(1000000) // Sleep for ~1000000 cycles, or 8ms for a 125MHz clock
```

2.6.7 Final Code

Here is the final code for this version of NW:

```
import spatial.dsl._
import org.virtualized._

object NW extends SpatialApp {
  @struct case class nw_tuple(score: Int16, ptr: Int16)

  @virtualize
  def main() {
    val SKIPB = 0 // move left
    val SKIPA = 1 // move up
    val ALIGN = 2 // move diagonal
    val MATCH_SCORE = 1
    val MISMATCH_SCORE = -1
    val GAP_SCORE = -1

    val seqA_text = loadCSV1D[MString]("/home/ChrisWunsch/seqA.txt", " ").apply(0) // Loads into␣
↪array with 1 string
    val seqB_text = loadCSV1D[MString]("/home/ChrisWunsch/seqB.txt", " ").apply(0) // Loads into␣
↪array with 1 string
    println("Aligning " + seqA_text + " and ")
    println("         " + seqB_text)

    val seqA_data = argon.lang.String.string2num(seqA_text) // Returns array of Int8
    val seqB_data = argon.lang.String.string2num(seqB_text) // Returns array of Int8

    val seq_length = seqA_data.length
    val LEN = ArgIn[Int]
    val LEN2x = ArgIn[Int]
    setArg(LEN, seq_length)
    setArg(LEN2x, seq_length*2)
    val seqA = DRAM[Int8](LEN)
    val seqB = DRAM[Int8](LEN)
    setMem(seqA, seqA_data)
    setMem(seqB, seqB_data)
    val seqA_aligned = DRAM[Int8](LEN)
    val seqB_aligned = DRAM[Int8](LEN2x)

    val max_length = 256
    val d = argon.lang.String.char2num("-")
    val dash = ArgIn[Int8]
    setArg(dash,d)
    val u = argon.lang.String.char2num("_")
    val underscore = ArgIn[Int8]
    setArg(underscore,u)
```

```
    Accel{
      val seqa_sram_raw = SRAM[Int8](max_length)
      val seqb_sram_raw = SRAM[Int8](max_length)

      seqa_sram_raw load seqA(0::LEN)
      seqb_sram_raw load seqB(0::LEN)

      val score_matrix = SRAM[nw_tuple](max_length+1,max_length+1)

      Foreach(LEN+1 by 1){ r =>
        Sequential.Foreach(0 until LEN+1 by 1) { c =>
          val previous_result = Reg[nw_tuple]
          val update = if (r == 0) (nw_tuple(-c.as[Int16], 0)) else if (c == 0) (nw_tuple(-r.
↪as[Int16], 1)) else {
            val match_score = mux(seqa_sram_raw(c-1) == seqb_sram_raw(r-1), MATCH_SCORE.
↪to[Int16], MISMATCH_SCORE.to[Int16])
            val from_top = score_matrix(r-1, c).score + GAP_SCORE
            val from_left = previous_result.score + GAP_SCORE
            val from_diag = score_matrix(r-1, c-1).score + match_score
            mux(from_left >= from_top && from_left >= from_diag, nw_tuple(from_left, SKIPB),␣
↪mux(from_top >= from_diag, nw_tuple(from_top,SKIPA), nw_tuple(from_diag, ALIGN)))
          }
          previous_result := update
          if (c >= 0) {score_matrix(r,c) = update}
        }
      }

      val traverseState = 0
      val padBothState = 1
      val doneState = 2

      val seqa_fifo_aligned = FIFO[Int8](max_length*2)
      val seqb_fifo_aligned = FIFO[Int8](max_length*2)
      val b_addr = Reg[Int](0)
      val a_addr = Reg[Int](0)
      b_addr := LEN
      a_addr := LEN
      val done_backtrack = Reg[Bit](false)
      FSM[Int](state => state != 2) { state =>
        if (state == traverseState) {
          if (score_matrix(b_addr,a_addr).ptr == ALIGN.to[Int16]) {
            seqa_fifo_aligned.enq(seqa_sram_raw(a_addr-1), !done_backtrack)
            seqb_fifo_aligned.enq(seqb_sram_raw(b_addr-1), !done_backtrack)
            done_backtrack := b_addr == 1.to[Int] || a_addr == 1.to[Int]
            b_addr :-= 1
            a_addr :-= 1
          } else if (score_matrix(b_addr,a_addr).ptr == SKIPA.to[Int16]) {
            seqb_fifo_aligned.enq(seqb_sram_raw(b_addr-1), !done_backtrack)
            seqa_fifo_aligned.enq(dash, !done_backtrack)
            done_backtrack := b_addr == 1.to[Int]
            b_addr :-= 1
          } else {
            seqa_fifo_aligned.enq(seqa_sram_raw(a_addr-1), !done_backtrack)
            seqb_fifo_aligned.enq(dash, !done_backtrack)
            done_backtrack := a_addr == 1.to[Int]
            a_addr :-= 1
          }
        } else if (state == padBothState) {
          seqa_fifo_aligned.enq(underscore, !seqa_fifo_aligned.full)
          seqb_fifo_aligned.enq(underscore, !seqb_fifo_aligned.full)
        } else {}
      } { state =>
        mux(state == traverseState && ((b_addr == 0.to[Int]) || (a_addr == 0.to[Int])),␣
↪padBothState,
          mux(seqa_fifo_aligned.full || seqb_fifo_aligned.full, doneState, state))
```

```
      }

      seqA_aligned(0::LEN2x) store seqa_fifo_aligned
      seqB_aligned(0::LEN2x) store seqb_fifo_aligned
    }

    val seqA_aligned_result = getMem(seqA_aligned)
    val seqB_aligned_result = getMem(seqB_aligned)

    val errors = seqA_aligned_result.zip(seqB_aligned_result){(a,b) => if (a != b && a != d && b␣
↪!= d) 1 else 0}.reduce{_+_}
    println("Found " + errors + " errors out of " + {seq_length*2} + " characters")
    val cksum = errors.to[Float] / (seq_length*2).to[Float] < 0.1.to[Float]
    println("Acceptable? " + cksum)

    val seqA_aligned_string = argon.lang.String.num2string(seqA_aligned_result)
    val seqB_aligned_string = argon.lang.String.num2string(seqB_aligned_result)
    println("Aligned A: " + seqA_aligned_string)
    println("Aligned B: " + seqB_aligned_string)
  }
}
```

2.7 5. Design Space Exploration with Spatial

Coming Soon!

- David

CHAPTER 3

Examples

This section gives some simple annotated applications for learning Spatial by example.

Please visit the latest regression branch of the spatial-apps repository: https://github.com/stanford-ppl/spatial-apps/tree/regression/src

CHAPTER 4

Targets

This section provides details on the hardware Spatial can target.

4.1 AWS Tutorial

This section describes how Spatial applications can be run on Amazon's EC2 FPGA instances. For an introduction to Spatial, see the tutorial in the *previous section*.

4.1.1 0. Introduction

Prerequisites

Running on EC2 FPGAs requires the following prerequisites:

- An installation of Spatial as described in the *previous tutorial*
- Vivado and a license to run on the desired Amazon FPGA. We tested this tutorial both on Amazon's FPGA Developer AMI which contains all the required software tools, as well as locally by following these instructions.

Setup

Clone Amazon's EC2 FPGA Hardware and Software Development Kit to any location:

```
git clone https://github.com/aws/aws-fpga.git
```

Spatial was most recently tested with version 1.3.3 of this repository (git commit 934000f9a57c0cde8786441864d5c6e0cf42fef9).

Set the `AWS_HOME` environment variable to point to the cloned directory. Also source the AWS setup scripts. The HDK script is needed for simulation and synthesis, and the SDK is needed to create the host binary:

```
export AWS_HOME=/path/to/aws-fpga
cd /path/to/aws-fpga/
source /path/to/aws-fpga/hdk_setup.sh
source /path/to/aws-fpga/sdk_setup.sh
```

For example, you can add the 4 commands above to your `.bashrc` and source that.

Finally, applications targeting the F1 board (in hardware or simulation) need to set the `target` variable. For example, make the following change in the very top of the `apps/src/MatMult_outer.scala` application:

```
object MatMult_outer extends SpatialApp {
  override val target = spatial.targets.AWS_F1  // <---- new line
  ...
```

The next tutorial sections describe how to generate and run applications for *simulation* and *for the FPGAs on the F1 instances*.

4.1.2 1. Simulation

Why Run Simulation

We intend for Spatial applications to work in hardware without requiring simulation. But for the alpha release of Spatial, the user may prefer to first simulate the design before paying to open an F1 FPGA instance. This tutorial describes how to simulate your Spatial application using the simulation environment provided by Amazon in their Development Kit.

These steps can be done on any machine with the Vivado XSIM simulator. We did this both on a local machine as well as on an EC2 machine with the Amazon FPGA Developer AMI. An FPGA instance is not needed for this simulation tutorial.

To skip simulation and run directly in hardware, see the *F1 tutorial*.

Generating the Application for Simulation

The first step is the same as compiling a Spatial application for any other target, shown here for the `SimpleTileLoadStore` example:

```
cd ${SPATIAL_HOME}
bin/spatial SimpleTileLoadStore --synth
```

You can replace `SimpleTileLoadStore` with any application, as described in *the previous tutorial*. Note however that the XSIM simulation seems to have a default timeout and that applications which run for too many cycles will not finish. When possible we recommend selecting input arguments which allow your design to complete more quickly in simulation.

Now generate the simulation binary `Top`:

```
cd ${SPATIAL_HOME}/gen/SimpleTileLoadStore
make aws-sim
./Top 100
```

Notice that the final two steps above both need the Vivado XSim simulator. Other simulators can be used with the Amazon Development Kit but this has not been tested with Spatial.

The simulation typically takes ten minutes or longer to complete.

If the simulation is successful, the following output will be seen:

```
PASS: 1
```

If your application completed successfully, next you can run it in *hardware*.

4.1.3 2. F1 Instances

Summary of Steps

Unlike simulation, running on the F1 requires a few simple manual steps. These depend on your personal AWS account (EC2 and S3). Specifically, following synthesis Amazon requires the bitstream to be uploaded to your S3 account, and an EC2 account is needed to launch an F1 instance to run the spatial application in hardware.

This tutorial describes the following steps:

- Authenticating your AWS account
- Generating and synthesizing a Spatial design. In our experience, synthesis/place/route takes 4-12 hours depending on design size
- Uploading the bitstream (AKA design checkpoint, or DCP) to Amazon S3 and waiting approximately 1 hour for the Amazon FPGA Image (AFI) associated with this bitstream to become valid
- Opening an F1 instance through your EC2 account
- Running the spatial application

Step 1: Authenticating your AWS account

Follow these steps to create a file `rootkey.csv`. This file can be placed anywhere, and will be needed for later steps to run commands associated with your AWS account.

Then add the path to that file to your .bashrc as follows:

```
export AWS_CONFIG_FILE=/path/to/rootkey.csv
```

Step 2: Generate and Synthesize your application

The first step is the same as compiling a Spatial application for any other target, shown here for the `MatMult_outer` example:

```
cd ${SPATIAL_HOME}
bin/spatial MatMult_outer --synth
```

You can replace `MatMult_outer` with any application, as described in *the previous tutorial*.

Now generate the F1 design and run synthesis/place/route to begin creating the Amazon FPGA Image (AFI):

```
cd ${SPATIAL_HOME}/gen/MatMult_outer
make aws-F1-afi
```

We tested this both using Amazon's FPGA Developer AMI and also locally using their instructions.

Notice that once this command completes, Vivado synthesis will be running in the background. A text file called `create_spatial_AFI_instructions.txt` has also been created. Follow the instructions in this text file (also described below) once Vivado completes to upload the Design Checkpoint (DCP) and finish creating the AFI.

Vivado will have completed once the `*.vivado.log` file in `aws-fpga/hdk/cl/examples/MatMult_outer/build/scripts` has printed `Finished creating final tar file in to_aws directory.`. You can also check if the `vivado` process has stopped.

Step 3: Creating the AFI

See the generated file `create_spatial_AFI_instructions.txt`, which guides the user step-by-step on how to upload the DCP to S3 and then run the `create-fpga-image` command.

Running these commands will require the AWS Command Line Interface. We tested with version 1.11.78.

Once `create-fpga-image` has been run, wait for the `logs` directory in S3 to be filled, and ensure that the AFI state in the file called `State` is "available".

Step 4: Opening an F1 instance

Start an F1 instance in the AWS console. We tested with Amazon's FPGA Developer AMI although this might not be necessary.

If you already have an existing F1 instance (e.g. for a previous Spatial application), skip to Step 5. If this is your first time starting the F1 instance, follow the one-time setup steps below.

Clone Amazon's EC2 FPGA Hardware and Software Development Kit to any location, e.g. `/home/centos/src/project_data`:

```
git clone https://github.com/aws/aws-fpga.git
```

Put the following (replacing with your chosen path above) in your `.bashrc`:

```
cd /home/centos/src/project_data/aws-fpga/
source /home/centos/src/project_data/aws-fpga/sdk_setup.sh
```

Source the `.bashrc`:

```
source ~/.bashrc
```

Then follow these instructions to build and install the required EDMA driver. Summary:

```
cd sdk/linux_kernel_drivers/edma/
make
echo 'edma' | sudo tee --append /etc/modules-load.d/edma.conf
sudo cp edma-drv.ko /lib/modules/`uname -r`/
sudo depmod
sudo modprobe edma-drv
```

Step 5: Running the Spatial application

Generate the host (software) binary using:

```
make aws-F1-sw
```

You can do this on your local machine and copy over the binary to the F1 (this might require changing permissions to run it), or compile the binary on the F1 instance. To do it on the F1 instance, you only need the `software/runtime` and `software/include` directories of the generated Spatial AWS application (e.g. `aws-fpga/hdk/cl/examples/MatMult_outer`), and can compile using `make all` in `software/runtime`.

Also modify the file `load.sh` in `software/runtime` to paste in the agfi ID returned above. Eventually this will be automated.

Run the application using the commands below in the `runtime` directory. Eventually the call to `load.sh` will be automated within `Top`:

```
bash load.sh
sudo ./Top arg1 arg2 ...
```

Notes on the above commands:

- Currently we require a board reset (part of `load.sh`) prior to running an application. Eventually this will not be needed.
- Eventually the `agfi` above will be automatically written to a file which the Spatial application reads. For now it is part of `load.sh`.

CHAPTER 5

Theory

This section provides a developer's introduction to the Spatial compiler to help you develop or modify the language. General users of the language can just look at the *tutorial*

5.1 Introduction

A Spatial program describes a dataflow graph consisting of various kinds of nodes connected to each other by data dependencies. Each node in a Spatial program corresponds to a architectural template. Spatial is represented in-memory as a parameterized, hierarchical dataflow graph.

Templates in Spatial capture parallelism, locality, and access pattern information at multiple levels. This dramatically simplifies coarse-grained pipelining and enables us to explicitly capture and represent a large space of designs which other tools cannot capture. Every template is parameterized. A specific hardware design point is instantiated from a Spatial description by instantiating all the templates in the design with concrete parameter values passed to the program. Spatial heavily uses metaprogramming, so these values are passed in as arguments to the Spatial program. The generated design instance is represented internally as a graph that can be analyzed to provide estimates of metrics such as area and cycle count. The parameters used to create the design instance can be automatically generated by a design space exploration tool.

(More to come)

CHAPTER 6

FAQ

6.1 Simulation

Why does my Scala result not match my RTL simulation result?

This is generally due to loop-carry dependency, pipelining, or parallelization issues. The best way to rule out these is to turn off all pipelining and parallelizations and check if the result is correct. If you have tried this and the results still do not match, it is possible you have exposed a compiler bug and should post to the Spatial users google group.

Why does my Scala result have a bunch of numbers with X's in them?

This means that this particular value was computed using uninitialized numbers at some point in its history.

6.2 Language

Why does Spatial use type classes rather than inheritance for things like Bits and Num?

While inheritance and type classes have many of the same properties, type classes are somewhat more flexible in that they allow simple conditional rules. Spatial views the idea that a type can be represented by a fixed set of Bits as a conditional property, rather than a superclass. Spatial uses type classes because this conditional attribution is not possible with normal inheritance.

For example, with the Tuple2[A,B] type, Spatial can conditionally determine if an instance of this type is representable by a fixed set of Bits based on the types A and B. If A and B both have the Bits property, then that particular instance will as well.

The downside of this is that type classes are based on type information only, not on the instance. This means, for example, that the VectorN type does not implicitly have evidence of being representable by Bits, even though every instance of that class will include a fixed width.

Why does Spatial include the VectorN type?

Unlike C++, Scala does not support template classes which are parameterized by integer values. This means that, for methods where the number of elements in the resulting Vector depends on an argument to

that method, the resulting type of the method cannot be written statically in Scala. However, this width is still statically analyzable. As a workaround, Spatial uses the VectorN type to denote a Vector which needs static analysis of its width.

If you could not find the answer to your question here, head over to the google group.

More to be added...

CHAPTER 7

Contact

Spatial is currently being developed in Professor Olukotun's group as part of the Stanford DAWN project.

Questions and issues can be directed either to the Spatial users google group or directly as an issue on the Spatial github .