Acceleration Tutorial

Loops and Pipelining

Version 2014.1

MAXELER
Technologies
MAXIMUM PERFORMANCE COMPUTING

Contents

Preface

Purpose of this document

This document covers features of MaxCompiler that allow you to implement loops and cyclic data flow within a Kernel, including transformation of software loops to pipelined implementations, and optimization of pipelined loops using loop unrolling, input transposition and loop tiling. These are key methods for exploiting application parallelism in MaxCompiler designs.

Each section introduces a new set of features and goes through examples showing their use, where appropriate.

Document Conventions

When important concepts are introduced for the first time, they appear in **bold**. *Italics* are used for emphasis. Directories and commands are displayed in `typewriter` font. Variable and function names are also displayed in `typewriter` font.

Java methods and classes are shown using the following format:

```
void optimization.pushPipeliningFactor(double pipelining)
```

C function prototypes are similar:

```
void RowSumLoopTile(
    int16_t  param_CFACTOR,
    int64_t  param_length,
   const float *instream_input,
   float  *outstream_output);
```

Actual Java usage is shown without italics:

```
carried sum <== stream.offset(new sum, -X);
```

C usage is similarly without italics:

```
sum[x] += input[count];
```

Sections of code taken from the source of the examples appear with a border and line numbers:

```
        tester .setKernelCycles(X * Y);
```

1 Introduction

The process of moving from a software implementation to a dataflow Kernel is often driven by the application's loop structure. The goal of this tutorial is to explain the various ways that loops in a software algorithm can be implemented as streaming, pipelined hardware.

We start with some very simple loops to see how they can be implemented in a Kernel and then turn to some more complex examples which include conditional loops and data dependency between loop iterations. We will cover a number of important techniques including loop unrolling, loop predication, input transposition and loop tiling, all of which can be applied to real-world Kernel implementations to great effect.

2 The Implicit Outer Loop

Consider the following software loop:

```
for (int count=0; ; count += 1) {
  B[count] = A[count] + 1;
}
```

When we implement this as a dataflow Kernel, the input array A will be streamed into the DFE (for example from the CPU memory to the DFE). Similarly, the output array B will be streamed out from the dataflow Kernel to the CPU.

Listing 1 shows the Kernel implementation for this trivial example. Looking at the body of the Kernel, we can see that there is no loop:

```
        // Input
        DFEVar input = io.input(”input”, dfeUInt(32));

        DFEVar result = input + 1;

        // Output
        io.output(”output”, result, dfeUInt(32));
```

The loop in the original software loop is a mechanism for addressing the data in the array. In a dataflow implementation, the data is streamed into the Kernel from the Manager, so there is no need for a loop in the Kernel itself: the loop is implicit in the stream.

The Manager controls the streaming of the data from its source, which can be from the CPU directly, from the LMem on the DFE or via MaxRing from another DFE. In this example, the data access is driven by the CPU code which triggers the copying of the A array to the DFE, and the copying of the B array back.

Listing 1: A trivial streaming Kernel (IncrementKernel.maxj).

```
/**
 * Document: Acceleration Tutorial - Loops and Pipelining (maxcompiler-loops.pdf)
 * Example: 1      Name: Increment
 * MaxFile name: Increment
 * Summary:
 *      Kernel design that increments the input stream and sends the
 *      result to the output stream.
 */
package increment;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class IncrementKernel extends Kernel {

    IncrementKernel(KernelParameters parameters) {
        super(parameters);

        // Input
        DFEVar input = io.input(”input”, dfeUInt(32));

        DFEVar result = input + 1;

        // Output
        io.output(”output”, result, dfeUInt(32));
    }
}
```

3 A Loop Counter

In this similar example, we need a variable `count` to track the loop control variable:

```
for (int count=0; ; count += 1) {
  output[count] = input[count] + count;
}
```

Listing 2 shows an implementation for this example. We still have no loop in the body of the Kernel, but we have added a counter to provide an iteration number for each data value presented on this input:

```
        DFEVar input = io.input(”input”, dfeUInt(32));

        DFEVar counter = control.count.simpleCounter(32);

        DFEVar result = input + counter;

        // Output
        io.output(”output”, result, dfeUInt(32));
```

Listing 2: A Kernel with a counter (SimpleCounterLoopKernel.maxj).

```
/**
 * Document: Acceleration Tutorial - Loops and Pipelining (maxcompiler-loops.pdf)
 * Example: 2      Name: Simple counter
 * MaxFile name: SimpleCounterLoop
 * Summary:
 *      A simple kernel that adds the value from a counter to a stream.
 */
package simplecounterloop;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class SimpleCounterLoopKernel extends Kernel {
    SimpleCounterLoopKernel(KernelParameters params) {
        super(params);

        // Input
        DFEVar input = io.input("input", dfeUInt(32));

        DFEVar counter = control.count.simpleCounter(32);

        DFEVar result = input + counter;

        // Output
        io.output("output", result, dfeUInt(32));
    }
}
```

4 A Loop Nest

Consider the following software loop nest:

```
int count = 0;
for (int y=0; y<Y; y++) {
  for (int x=0; x<X; x++) {
    output[count] = input[count]+(y*100)+x;
    count += 1;
  }
}
```

Listing 3 shows a Kernel that implements this design.

The body of the Kernel now includes a counter chain to provide a pair of indices for each data value presented on this input:

```
        // Set up counters for 2D loop
        CounterChain chain = control.count.makeCounterChain();
        DFEVar y = chain.addCounter(Y, 1).cast(dfeUInt(32));
        DFEVar x = chain.addCounter(X, 1).cast(dfeUInt(32));

        DFEVar result = input + (y * 100) + x;
```

The loop body is executed for every input value. The `count` variable in the original software source is implicit in the stream.

Listing 3: A Kernel with counters to track a 2D loop nest (Simple2dCounterKernel.maxj).

```
/**
 * Document: Acceleration Tutorial - Loops and Pipelining (maxcompiler-loops.pdf)
 * Example: 3      Name: Simple two-dimensional counter
 * MaxFile name: Simple2dCounter
 * Summary:
 *      Kernel design that implements a nested loop computation using a counter
 *      chain to provide a pair of indices for each data value presented on the
 *      input.
 */
package simple2dcounter;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.CounterChain;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class Simple2dCounterKernel extends Kernel {

    Simple2dCounterKernel(KernelParameters params, int X, int Y) {
        super(params);

        // Input
        DFEVar input = io.input("input", dfeUInt(32));

        // Set up counters for 2D loop
        CounterChain chain = control.count.makeCounterChain();
        DFEVar y = chain.addCounter(Y, 1).cast(dfeUInt(32));
        DFEVar x = chain.addCounter(X, 1).cast(dfeUInt(32));

        DFEVar result = input + (y * 100) + x;

        // Output
        io.output("output", result, dfeUInt(32));
    }
}
```

5 Data Dependency

Consider the following software loop nest, which implements a Newton-Raphson reciprocal approximation for $0.5 < d < 1.0$, where four iterations are enough for 32-bit accuracy:

```
for (count=0; ; count += 1) {
  float d = input[count];
  float v = 2.9142 - 2*d;
  for (iteration=0; iteration < 4; iteration += 1) {
    v = v * (2.0 - d * v);
  }
  output[count] = v;
}
```

In the trivial loop examples we have looked at so far, each iteration of the loop relied only on the values in the incoming stream and the loop counters. In this loop, however, each iteration of the loop depends on the value of `v` calculated in the previous iteration.

Listing 4: Computing a reciprocal using the Newton-Raphson method (ReciprocalKernel.maxj).

```
/**
 * Document: Acceleration Tutorial - Loops and Pipelining (maxcompiler-loops.pdf)
 * Example: 4      Name: Reciprocal
 * MaxFile name: Reciprocal
 * Summary:
 *      Kernel design that implements a reciprocal using the Newton-Raphson
 *      method.
 */
package reciprocal;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class ReciprocalKernel extends Kernel {

    ReciprocalKernel(KernelParameters params) {
        super(params);

        // Input
        DFEVar d = io.input(”d”, dfeFloat(8, 24));

        DFEVar v = 2.9142 - 2.0*d;

        for (int iteration = 0; iteration < 4; iteration += 1) {
            v = v*(2.0 - d*v);
        }

        // Output
        io.output(”output”, v, dfeFloat(8, 24));
    }
}
```

6 Loop Unrolling

In this case, we can *unroll* the inner loop and create four copies of the hardware that constitutes its body. This creates a pipeline where each stage of the pipeline calculates a value of `v` based on the value of `v` from the previous stage of the pipeline. The pipeline still produces one result per tick, but it takes a number of ticks for the result for a given input to propagate to the output. This is called the *latency* of the pipeline.

Listing 4 shows a Kernel implementation of the unrolled loop. To unroll the loop we use a Java loop to generate the data path at *construction time*:

```
        for (int iteration = 0; iteration < 4; iteration += 1) {
            v = v*(2.0 - d*v);
        }
```

7 Quantifying pipeline parallelism

The resultant Kernel graph is shown in *Figure 1*. Notice that there is no control element to this graph: the implementation is purely data path. The values of `v` passed from stage to stage are labeled in the graph.

This design uses replicated hardware (for each of the four loop iterations) in order to be able to deliver one result per tick. Since each iteration itself involves two deeply-pipelined floating-point multiplies (about 13 ticks each) and one subtract (about 12 ticks each), the total amount of parallelism is quite large — around $(2 \times 13 + 12) \times 4 + 2 = 154$-fold. Because there is a dependency between all of the operations, this is also the pipeline depth of the graph.

In the case of acyclic dataflow graphs like this one, the depth of the pipeline is a concern only from a resource utilization point of view, provided the number of values to process is much greater than the pipeline latency. The depth of the pipeline means that the first result has a latency of a number of ticks, but even a latency of thousands of ticks has no significance in most streaming applications.

In *section 11* we consider cyclic graphs, where the pipeline latency becomes a correctness concern.

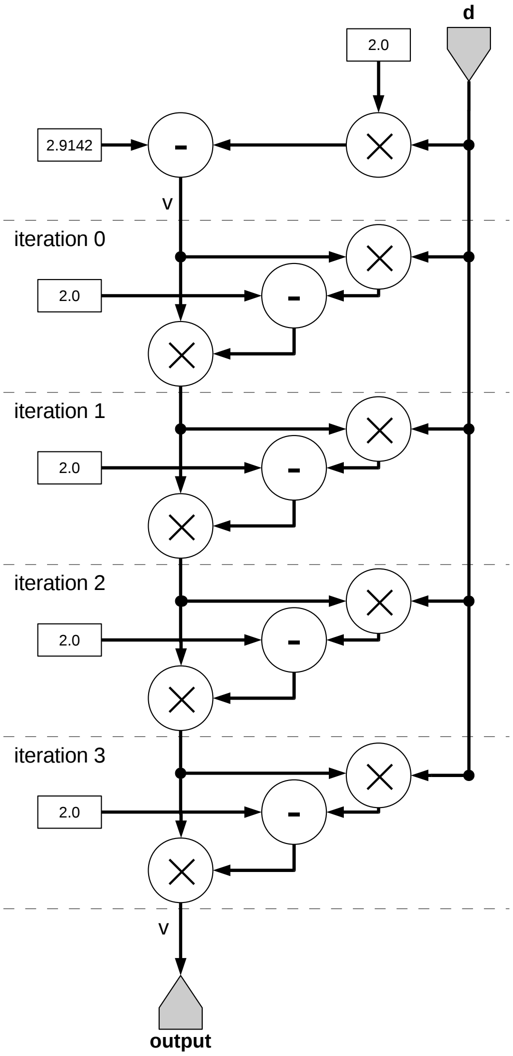


Figure 1: Kernel graph for an unrolled Newton-Raphson reciprocal approximation

8 Predicating a While Loop

Consider now a loop with a while loop inside, such as this example, which finds the rightmost occurrence of the 10-bit binary sequence 1010010001 (= 0x291) in a 32-bit input, returning -1 if the pattern is not found:

```
for (count=0; ; count += 1) {
  int d = input[count];
  int result = 0;
  int found = 0;
  while (d != 0 && result < 22 &! found) {
    if ((d & 0x3FF) != 0x291)
      result = result + 1;
    else
      found = 1;
    d = d >> 1;
  }
  result = found ? result : -1;
  output[count] = result;
}
```

In software, we want to keep the number of iterations required to a minimum, so we regularly use while loops or breaks in for loops to jump out of a loop. In our heavily-pipelined dataflow environment, such dynamic control flow would break our regular pipelines, so we must re-implement such loops in another way.

In this case we know the while loop can execute at most $32 - 10 = 22$ times, so we can avoid dynamic control flow by replacing it with a for loop. The difficulty is to extract the correct final value for the result variable. Here is one solution, presented as a software implementation:

```
for (count=0; ; count += 1) {
  int d = input[count];
  int result = 0;
  int found = 0;
  for (int i = 0; i < 22; ++i) {
    int condition = (d & 0x3FF) == 0x291;
    found = condition ? 1 : found;
    result = found ? result : result + 1;
    d = d >> 1;
  }
  result = found ? result : -1;
  output[count] = result;
}
```

The Boolean variable found is set true the first time condition is true. We use this to *predicate* subsequent updates to result. The inner loop now always executes for 22 iterations. We can also remove the test for d being 0 as we cannot shorten the number of iterations. At the end of the loop, we can set the output to -1 if the pattern has not been found in d.

We can use this idea to generate hardware by unrolling the loop as we did in *section 6*, as shown in *Listing 5*.

Listing 5: Class for the bit-pattern search Kernel (BitsearchKernel.maxj).

```
/**
 * Document: Acceleration Tutorial - Loops and Pipelining (maxcompiler-loops.pdf)
 * Example: 5      Name: Bitsearch
 * MaxFile name: Bitsearch
 * Summary:
 *      Kernel design that which finds the rightmost occurrence of the 10-bit
 *      binary sequence 1010010001 (= 0x291) in a 32-bit input, returning -1 if
 *      the pattern is not found.
 */
package bitsearch;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class BitsearchKernel extends Kernel {

    BitsearchKernel(KernelParameters parameters) {
        super(parameters);

        // Input
        DFEVar d = io.input(”d”, dfeUInt(32));

        DFEVar result = constant.var(dfeInt(5), 0);
        DFEVar found = constant.var(dfeBool(), 0);

        for (int i = 0; i < 22; ++i) {
            DFEVar condition = ((d & 0x3FF) === 0x291);
            found = condition ? 1 : found;
            result = found ? result : result + 1;
            d = d >> 1;
        }
        result = found ? result : -1;
        // Output
        io.output(”output”, result.cast(dfeInt(32)), dfeInt(32));
    }
}
```

Figure 2 shows the Kernel graph for this implementation. In the body of the Kernel, we generate 22 copies of the body of the loop, with the variable `finished` passed through the pipeline from stage to stage, along with the result `result`:

```
        DFEVar result = constant.var(dfeInt(5), 0);
        DFEVar found = constant.var(dfeBool(), 0);

        for (int i = 0; i < 22; ++i) {
            DFEVar condition = ((d & 0x3FF) === 0x291);
            found = condition ? 1 : found;
            result = found ? result : result + 1;
            d = d >> 1;
        }
        result = found ? result : -1;
```

Notice that in this case, we are not just replicating the data path within the loop, but also all of the control logic for the predication.

A `for` loop with a break can be treated similarly to a `while` loop.

9 Loop Unrolling and Logic Utilization

Whilst loop unrolling achieves our throughput goal of one datum per tick, it does so at the expense of logic utilization, where the loop body logic is replicated for every iteration. For a simple loop which does not need predication, the replicated pipeline stages are working every tick, giving us full pipeline utilization, but with a predicated loop, each stage may not be utilized fully, or indeed very frequently at all.

In the original software implementation, the loop was executed the number of times required for each datum; in the unrolled implementation, we need to unroll the loop to the maximum number of iterations. If the calculation for a datum passing through the unrolled logic is actually complete before it has reached the end of the pipeline, it still passes through all of the remaining stages but no computation is performed on that datum in these stages, meaning that they are not doing anything useful for that tick.

The percentage of the time that the pipeline stages are utilized depends on the data itself. If the data infrequently uses some of the later pipeline stages, then the implementation can often be re-factored to be *partially unrolled*, such that the average throughput for a real data set is one result per stream tick. Partial loop unrolling introduces cyclic data flow, which we will cover in the next section.

This can result in (often significantly) improved logic utilization and increased overall throughput on larger designs (for example with multiple pipelines per DFE). It is important to use realistic data sets when making trade-offs between logic utilization and performance as an unrepresentative set of data can lead to a sub-optimal solution.

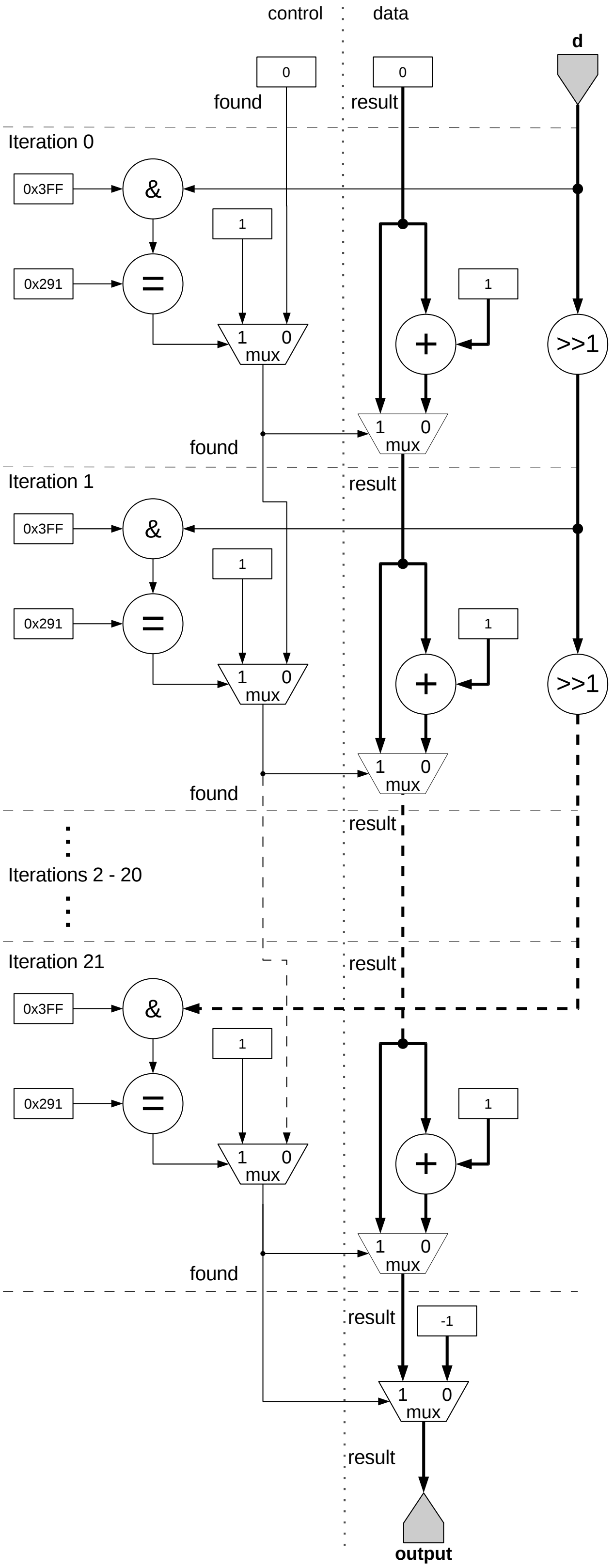


Figure 2: Kernel graph for the unrolled sequence matching Kernel

10 Cyclic Data Flow

All the loops we have seen so far have resulted in *acyclic* data-flow graphs i.e. graphs where the data flows in one direction, from inputs to outputs, with no loops going back up the graph.

A cycle in the data-flow graph arises from a dependence from one iteration to the next: a *loop-carried dependence*. Consider this software example:

```
int count = 0;

for (int y=0; ; y += 1) {
  sum[y] = 0.0;
  for (int x=0; x<X; x += 1) {
    sum[y] = sum[y] + input[count]
    count += 1;
  }
  output[y] = sum[y];
}
```

This accepts as its input an $X \times Y$ array and computes the sum of each row. *Figure 3* shows the array with its dimensions, the direction of the loop variables and the coordinates of the corners for clarity.

The array data is streamed into the Kernel graph row-by-row. We aim to produce, as output, one sum value for each row.

The inner loop has a carried dependence due to `sum[y]`. One option might be to remove the cycle by fully-unrolling the innermost loop (as described in *section 6*). However, if `X` is large or the loop body complex, this creates a prohibitively large amount of logic: loop unrolling worked well for the loop-carried dependence in the Newton-Raphson reciprocal iteration in *section 6* since just four iterations were sufficient.

We would like to build a data path that passes the `sum[y]` back into the adder, creating a cycle in the graph. A Kernel graph showing how we might consider implementing this design is shown in *Figure 4*. The backward edge that we want to create is shown as a dotted red line.

The code intended to generate this design is shown in *Listing 6*. We use a simple counter `x` to keep track of the current position in the row:

```
        DFEVar x = control.count.simpleCounter(MathUtils.bitsToRepresent(X), X);
```

We then create a variable `carriedSum` that carries the value from the previous iteration:

```
        DFEVar carriedSum = scalarType.newInstance(this);
```

`carriedSum` is created using `newInstance(this)`, which makes a *source-less* stream: we will connect its source shortly.

The loop consists of a head, a body and a foot. The head has a multiplexer that selects the initial value 0.0 on iteration 0, i.e. at the start of each row, then takes the carried value for subsequent iterations:

```
        DFEVar sum = x === 0 ? 0.0 : carriedSum;
```

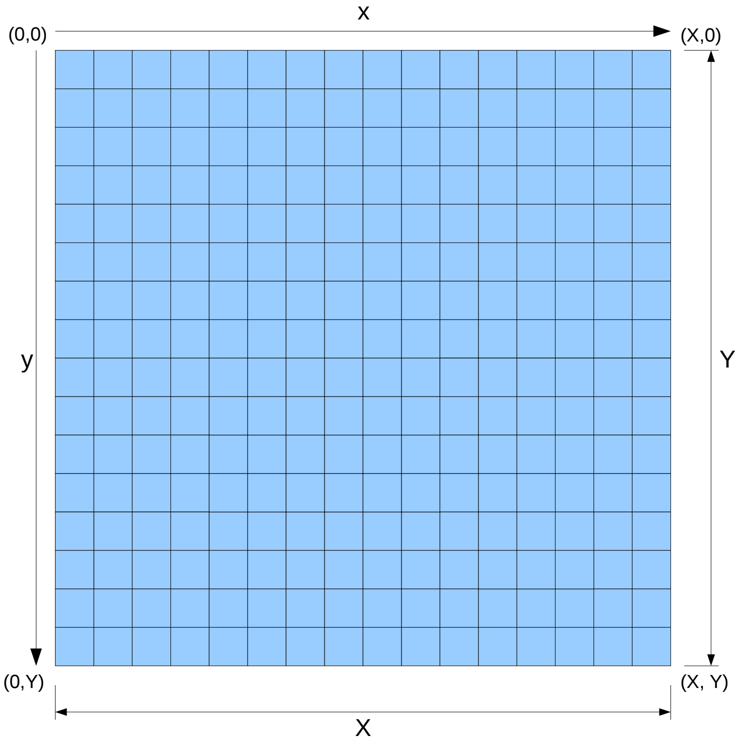


Figure 3: Diagram of an input array for the row-sum application; each cell contains a number, and the desired output is a sequence of numbers, each being the sum of the numbers in one row of the array

The body computes the new value of the sum:

```
        DFEVar newSum = input + sum;
```

The foot is where we use `carriedSum <== X` to connect the new sum back to the head. <== is the *connect* operator, and `x<==y` is equivalent to `x.connect(y)`. An offset (`stream.offset(newSum, -1)`) refers to the value of `newSum` from the previous iteration:

```
        carriedSum <== stream.offset(newSum, -1); // scheduling fails
```

Finally, we would like the sum for each row to be written into the output stream. We include a control parameter here, which enables the output stream just at the end of each row:

```
        io.output(”output”,  newSum, scalarType, x === (X - 1));
```

However, when we build this design, the Kernel Compiler complains that the design cannot be scheduled:

```
Thu 13:09: ERROR:   Found illegal loop with a latency of 12 (>0)!
```

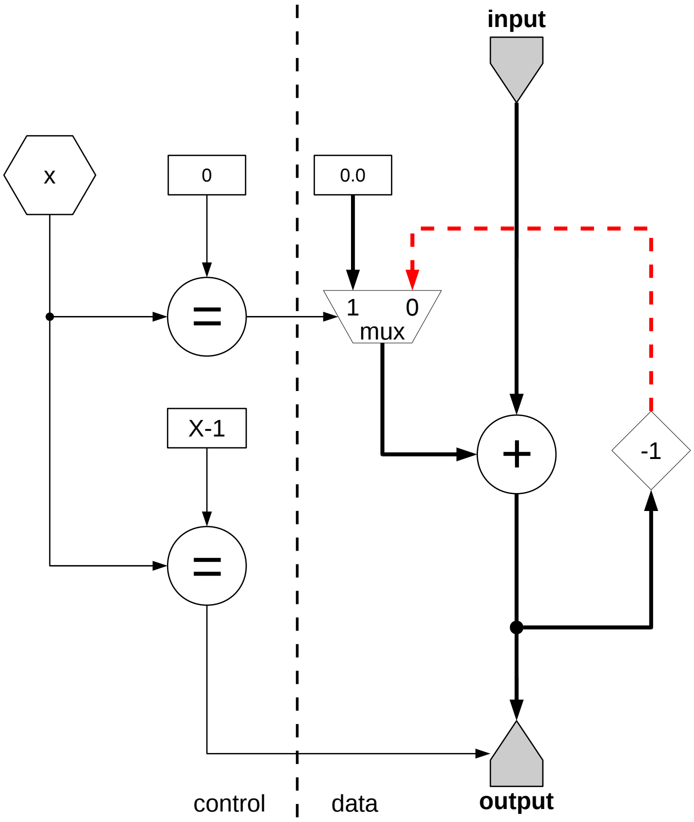


Figure 4: Kernel graph for a potential implementation for the row sum problem

Listing 6: Class for the row-wise summation Kernel (RowSumIncorrectKernel.maxj).

```
/**
 * Document: Acceleration Tutorial - Loops and Pipelining (maxcompiler-loops.pdf)
 * Example: 6      Name: Rowsum incorrect
 * MaxFile name: RowSumIncorrect
 * Summary:
 *      Incorrect kernel design for the row sum problem.
 *  THIS EXAMPLE DOES NOT BUILD!
 */
package rowsumincorrect;

import com.maxeler.maxcompiler.v0.utils.MathUtils;
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

/****************************************************************
* IMPORTANT: THIS EXAMPLE DOES NOT BUILD!
 ****************************************************************/
class RowSumIncorrectKernel extends Kernel {

    final DFEType scalarType = dfeFloat(8, 24);

    RowSumIncorrectKernel(KernelParameters parameters, int X) {
        super(parameters);

        // Input
        DFEVar input = io.input(”input”, scalarType);

        // Set up counter
        DFEVar x = control.count.simpleCounter(MathUtils.bitsToRepresent(X), X);

        // At the head of the loop, we select whether to take the initial value
        // or the value that is being carried around the data-flow cycle
        DFEVar carriedSum = scalarType.newInstance(this);
        DFEVar sum = x === 0 ? 0.0 : carriedSum;

        // The loop body itself
        DFEVar newSum = input + sum;

        // At the foot of the loop, we add the backward edge
        carriedSum <== stream.offset(newSum, -1); // scheduling fails

        // Changing the line above to this would build but produce incorrect results.
        // carried_sum <== stream.offset(new_sum, -13);

        // We have a controlled output to deliver the sum at the end of each row
        io.output(”output”, newSum, scalarType, x === (X - 1));
    }
}
/****************************************************************
 * IMPORTANT: THIS EXAMPLE DOES NOT BUILD!
 ****************************************************************/
```

11 Pipeline Depth

Understanding the scheduling failure requires attention to details we have previously ignored. MaxCompiler creates deep pipelines to get maximum performance, so each of the operations can take multiple clock ticks to produce an output. Previous examples have required only acyclic Kernel graphs, for which MaxCompiler can organize the flow of input data automatically to each operating unit at the correct time. With cyclic dataflow, organizing the data is non-obvious and becomes the designer's responsibility to a greater extent. When deciding how far downstream to tap the pipeline, the designer must consider not only the latency of the intervening stages but the semantic implications.

In our example, the floating-point adder is a multi-stage pipeline. The result is available 12 ticks after the operation starts, and the multiplexer before the adder has a pipeline depth of 1, so the total depth for our pipeline is 13. Setting the stream offset to -13 or a more negative number would allow MaxCompiler to schedule the design successfully. That is, changing the line:

```
carriedSum <== stream.offset(newSum, -1);
```

in *Listing 6* to:

```
carriedSum <== stream.offset(newSum, -13);
```

prevents the error message and allows the build to proceed.

Although modifying the stream offset in this way addresses the issue of latency, two problems remain:

- Knowing the pipeline depth for this particular example is no help in general, because it will differ in other applications depending on their particular component latencies, which are not guaranteed.
- More significantly, this change does not give the correct mathematical result.

With regard to the mathematical result, the number arriving on each tick is not added to the cumulative sum of all preceding numbers in the row as it should be, but to the sum of a combination of leftover pipeline contents and any numbers having arrived at least 13 ticks earlier, *which produces incorrect results*. The adder needs to read a new input only when there is a valid cumulative partial sum coming from the stream offset, which is 13 ticks later. One way to do this is by adding an extra counter to control the input and output, as explained in *section 13*.

As for the problem of creating a valid, minimal offset, we can let MaxCompiler choose a sufficient loop offset automatically instead of attempting to estimate and code it manually.

12 Autoloop Offsets

An **Autoloop Offset** is a feature of MaxCompiler that automatically calculates the lowest valid offset for a cycle in the graph.

12.1 AutoLoop Offset Initialization

To let MaxCompiler choose a stream offset automatically, we must declare a variable of type *OffsetExpr* to plug into the *stream.offset()* method, instead of using a hard-coded constant. Rather than initializing the variable with a known value, we initialize it using one of these methods:

*Stream.OffsetExpr makeOffsetAutoLoop(**String** name)*

*Stream.OffsetExpr makeOffsetAutoLoop(**String** name, **int** min_size, **int** max_size)*

*Stream.OffsetExpr makeOffsetAutoLoop(**String** name, OffsetExpr min_size, OffsetExpr max_size)*

The *name* parameter uniquely identifies the offset expression within the Kernel name space. The optional minimum and maximum size parameters constrain the value to a preferred range.

These methods instruct MaxCompiler to infer a value for the variable based on how it is used elsewhere in the code. If the variable is used anywhere in a *stream.offset()* method to define the offset of a stream, MaxCompiler tries to infer the minimum positive value for it that allows scheduling to succeed. If no such value exists, or if none exists within the range specified by the minimum and maximum parameters, scheduling fails.

> ✷ MaxCompiler computes the smallest possible offset value by default, so specifying the minimum and maximum values is useful only for expressing application-specific scheduling constraints.

12.2 AutoLoop Offset Usage

Normally it is best to use a separate AutoLoop offset for each stream that needs one. If an AutoLoop offset variable is reused, MaxCompiler has to infer a value constrained by every usage simultaneously: scheduling may fail unless one size fits all cases, where it might have succeeded if a variety of offsets had been allowed. Unnecessary minimum and maximum constraints should be avoided for the same reason.

The value inferred for the AutoLoop offset is always positive, whereas the stream offset for any backward edge in a graph is negative. The parameter passed to the *stream.offset()* method in this case is most typically the negation of an AutoLoop offset variable rather than the variable itself.

> ✷ It is easy to forget to negate the Autoloop offset value when passing it to a stream offset: verifying this should be the first step to troubleshoot scheduling issues.

12.3 AutoLoop Offset Evaluation

It is often necessary to use an AutoLoop offset value in calculations or comparisons. Expressions of the form x`.getDFEVar(...)` with an AutoLoop offset x and one of the following methods are helpful for this purpose.

DFEVar Stream.OffsetExpr.getDFEVar(KernelLib design)

DFEVar Stream.OffsetExpr.getDFEVar(KernelLib design, DFEType type)

The type parameter is optional if the AutoLoop offset has been defined with a minimum and maximum size, and required if it has not. When the type parameter is given, it must be of sufficient bit width to represent the offset. For example, this code takes an AutoLoop offset `someAutoLoop` to an unsigned eight bit variable and calls it `someDFEVar`:

```
DFEVar someDFEVar = someAutoLoop.getDFEVar(this,dfeUInt(8));
```

CPU applications or Manager code for calculating mode parameters may also need to have the value of an AutoLoop offset as an integer type, for example to determine the number of Kernel ticks to run for.

12.4 AutoLoop Offsets and Stream Offset Parameters

Offset expressions containing AutoLoop offsets may also contain variable stream offset parameters that are created by the `stream.makeOffsetParam()` method and assigned by the CPU application, as described in Section 5.3 of the MaxCompiler Kernel Tutorial. In an offset expression such as $a + n$, where a is an AutoLoop offset and n is an offset parameter, the value MaxCompiler infers for a depends on the value the CPU application assigns to n. A larger value for n enables a smaller value for a.

If an AutoLoop offset referenced in a call to *max_get_offset_auto_loop_size()* depends on a variable stream offset parameter in this sense, then the CPU application must set it before calling the function. Enforcement of this condition is not guaranteed by MaxCompiler, but a violation is likely to cause an incorrect return value.

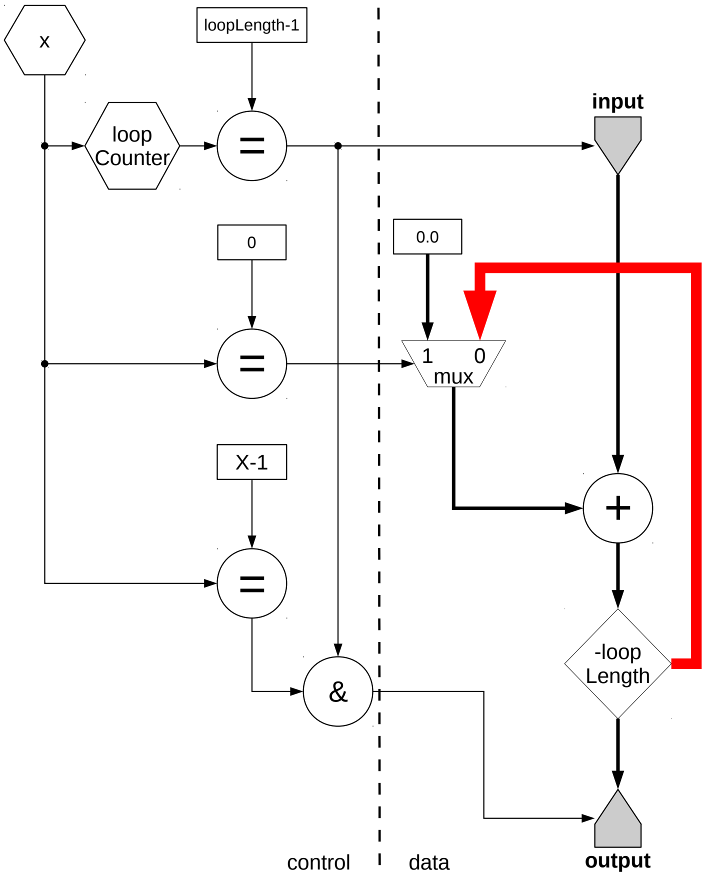


Figure 5: Kernel graph for a multi-tick implementation of the row sum problem

13 Multi-tick Implementation

AutoLoop offsets allow us to revisit the Row Sum application example and develop a correct solution. *Figure 5* shows the Kernel graph for this design. The loop coming out of the stream offset is shown as a heavy red line. The code for the Kernel is shown in *Listing 7*.

In this example, we have added an extra counter, chained to `x`. We set this counter to count up to a value `loopLength` that is initialized by MaxCompiler to be large enough to schedule the pipeline:

```
        OffsetExpr loopLength = stream.makeOffsetAutoLoop("loopLength");
        DFEVar loopLengthVal = loopLength.getDFEVar(this, dfeUInt(8));
        CounterChain chain = control.count.makeCounterChain();
        DFEVar x = chain.addCounter(X, 1);
        DFEVar loopCounter = chain.addCounter(loopLengthVal, 1);
```

We only read a new input from the input stream once every `loopLength` ticks:

```
        DFEVar input = io.input("input", scalarType, loopCounter === (loopLengthVal-1));
```

Listing 7: Kernel for a multi-tick implementation of the row sum problem using AutoLoop offsets (AutoRowSumKernel.maxj).

```
/**
 * Document: Acceleration Tutorial - Loops and Pipelining (maxcompiler-loops.pdf)
 * Example: 7      Name: Automatic row-sum
 * MaxFile name: AutoRowSum
 * Summary:
 *      Kernel design for a multi-cycle implementation of the row sum problem
 *      using auto loop offsets.
 */

package autorowsum;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.CounterChain;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Stream.OffsetExpr;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class AutoRowSumKernel extends Kernel {

    // final DFEType scalarType = dfeUInt(32);
    final DFEType scalarType = dfeFloat(8, 24);

    AutoRowSumKernel(KernelParameters parameters, int X) {
        super(parameters);

        // Set up counters for 2D loop
        OffsetExpr loopLength = stream.makeOffsetAutoLoop(”loopLength”);
        DFEVar loopLengthVal = loopLength.getDFEVar(this, dfeUInt(8));
        CounterChain chain = control.count.makeCounterChain();
        DFEVar x = chain.addCounter(X, 1);
        DFEVar loopCounter = chain.addCounter(loopLengthVal, 1);

        // Input
        DFEVar input = io.input(”input”, scalarType, loopCounter === (loopLengthVal-1));

        // The loop body itself
        // At the head of the loop, we select whether to take the initial value,
        // or the value that is being carried around the loop cycle
        DFEVar carriedSum = scalarType.newInstance(this); // sourceless stream
        DFEVar sum = x === 0 ? 0.0 : carriedSum;

        DFEVar newSum = input + sum;

        DFEVar newSumOffset = stream.offset(newSum, -loopLength);

        // At the foot of the loop, we add the backward edge
        carriedSum <== newSumOffset;

        // We have a controlled output to deliver the sum at the end of each row
        io.output(”output”, newSum, scalarType, x === (X - 1) & loopCounter === (loopLengthVal-1));
    }
}
```

We connect a backward edge to the input of the adder using a stream offset of `loopLength`:

```
        // At the head of the loop, we select whether to take the initial value,
        // or the value that is being carried around the loop cycle
        DFEVar carriedSum = scalarType.newInstance(this); // sourceless stream
        DFEVar sum = x === 0 ? 0.0 : carriedSum;

        DFEVar newSum = input + sum;

        DFEVar newSumOffset = stream.offset(newSum, -loopLength);

        // At the foot of the loop, we add the backward edge
        carriedSum <== newSumOffset;
```

Finally, the output is only enabled at the end of every row when the output of the adder is valid:

```
        io.output(”output”, newSum, scalarType, x === (X - 1) & loopCounter === (loopLengthVal-1));
```

We do not need to be precise about the actual size of our offset. Correct operation depends only on outputting the same offset value as we use to loop back to the input of our adder. Clearly, a long pipeline delay uses extra resources, but if there were a need to increase it (by passing a minimum value to the `makeOffsetAutoLoop()` method), doing so would not break the design.

Note that the number of ticks that the Kernel is run for must be multiplied by `loopLength`. In this example, it is done in the default mode of the Manager:

```
        InterfaceParam length = ei.addParam(”length”, CPUTypes.INT);
        InterfaceParam lengthInBytes = length ∗ CPUTypes.FLOAT.sizeInBytes();
        InterfaceParam loopLength = ei.getAutoLoopOffset(”AutoRowSumKernel”, ”loopLength”);
        ei.ignoreAutoLoopOffset(”AutoRowSumKernel”, ”loopLength”);

        ei.setTicks(”AutoRowSumKernel”, length ∗ loopLength);
```

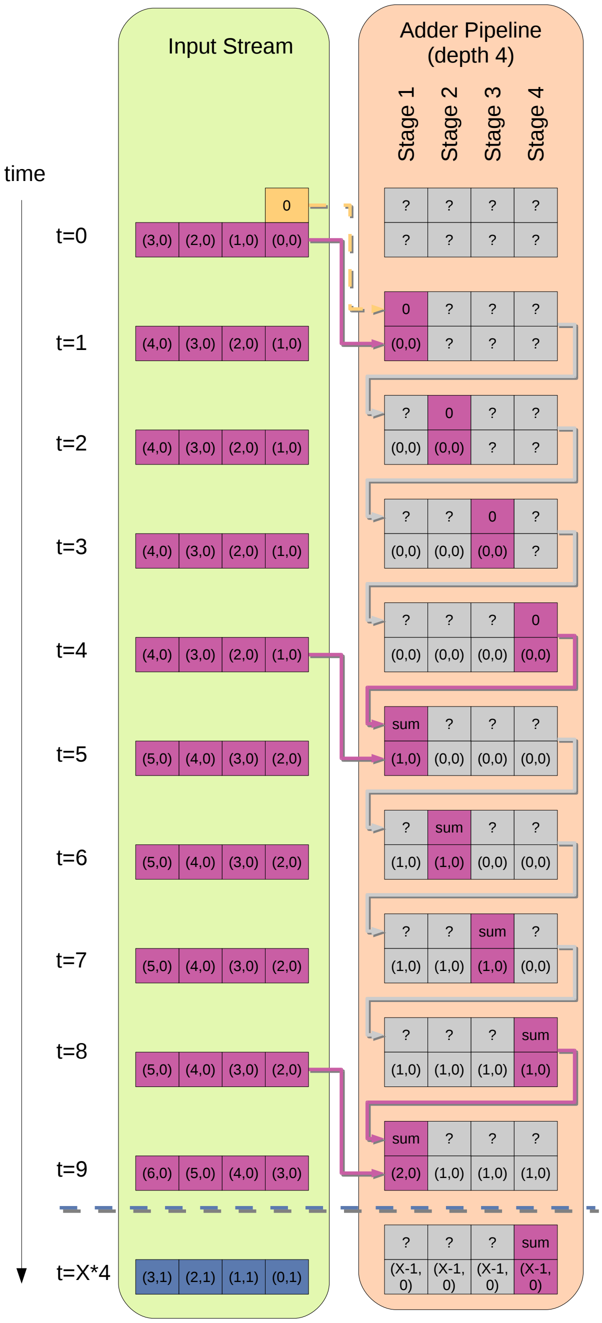
This uses the method `mode.getAutoLoopOffset`, which gets the value of the auto loop offset and adds a parameter to the mode. In this example, the CPU code does not need to know the value of the offset, so the method `mode.ignoreAutoLoopOffset` is used to prevent the offset parameter appearing in the SLiC API for this `.max` file.

We now have a multi-tick implementation of the row-sum problem, but we have taken a big performance hit: we can now only do an add every `loopLength` ticks.

Figure 6 shows how the adder pipeline looks over time. To simplify the explanation, the multiplexer-adder pipeline depth is reduced to 4:

- In the initial state (`t=0`), the pipeline is uninitialized, so we represent that with question marks.
- In the next tick,(`t=1`), we put the constant `0` and the first element from the first row, `(0,0)` into the adder pipeline.
- In the third, (`t=2`), fourth, (`t=3`), and fifth (`t=4`) ticks, the addition for the constant `0.0` and co-ordinate `(1,0)` propagate through the pipeline. Rather than show a partial sum for each stage of the adder pipeline, we represent these partial sums with the original arguments propagating through the pipeline. We need to pause the input stream until we can read in another value when the output from the adder is valid.
- In the sixth tick, (`t=5`), the result of our first add, `sum` is now available, so we can read in a new value from the input stream, element `(1,0)`, and add it to this result.
- This loop continues, doing a new add every 4 ticks, until we hit the end of the line and we can output a result.

From the gray stages of the adder pipeline in the diagram, it is clear that the adder pipeline is not used for any useful calculation three quarters of the time. In our real implementation with a pipeline depth of 13 for the adder and multiplexer, things are even worse, with the adder only doing something useful 1 in every 13 ticks. To get the maximum performance out of the dataflow engine, we need to put a new set of operands into the adder pipeline every tick so that we can get a throughput of one (*useful*) add per tick.



The pipeline is empty.

0 is read in as the initial state of the sum.
The first item is read from the input stream.
The adder starts computation of the first sum

The input is paused.
The first add is propagating through the adder.

The input is paused.
The first add is propagating through the adder.

The input is paused.
The first add is propagating through the adder.

The first add is complete.
The sum is fed back into the start of the adder.
The next item is read from the input stream.
The adder starts computation of the second add.

The input is paused.
The second add is propagating through the adder.

The input is paused.
The second add is propagating through the adder.

The input is paused.
The second add is propagating through the adder.

The second add is complete.
The sum is fed back into the start of the adder.
The next item is read from the input stream.
The adder starts computation of the third add.

The result of the first row sum is complete.

Figure 6: Underused pipeline in the multi-tick implementation of the row-sum

Listing 8: CPU code for the row sum Kernel (AutoRowSumCpuCode.c).

```
 */
#include <stdlib.h>
#include <stdint.h>

#include <MaxSLiCInterface.h>
#include ”Maxfiles.h”

const int X = AutoRowSum_X; // image row length (number of columns)
const int Y = 64;  //  image column length (number of rows)

void generateInputData(float *input)
{
    for (int y = 0; y < Y; ++y)
        for (int x = 0; x < X; ++x)
            input[y * X + x] = (y + x);
}

void AutoRowSumCPU(float *input, float *output)
{
    for (int y = 0; y < Y; ++y) {
        output[y] = 0.0;
        for (int x = 0; x < X; ++x)
            output[y] += input[y * X + x];
    }
}

int check(float *output, float *expected)
{
    int status = 0;
    for (int i = 0; i < Y; i++) {
        if (output[i] != expected[i]) {
             fprintf (stderr, ”[%d] error, output: %f != expected: %f\n”,
                 i, output[i], expected[i]);
            status = 1;
        }
    }
    return status;
}

int main()
{
    const int size = X * Y;
    int sizeBytes = size * sizeof(float);
    float *input = malloc(sizeBytes);
    float *output = malloc(sizeBytes);
    float *expected = malloc(sizeBytes);

    generateInputData(input);

     printf (”Running DFE.\n”);
    AutoRowSum(size, input, output);

    AutoRowSumCPU(input, expected);

    int status = check(output, expected);
    if (status)
        printf (”Test failed.\n”);
    else
        printf (”Test passed OK!\n”);

    return status;
}
```

14 Loop-carried Dependence Distance

Consider again our row-sum example. Suppose that instead of summing the rows, we wanted to sum the *columns*. If our data were in the same order, in sketch form we would want something like this:

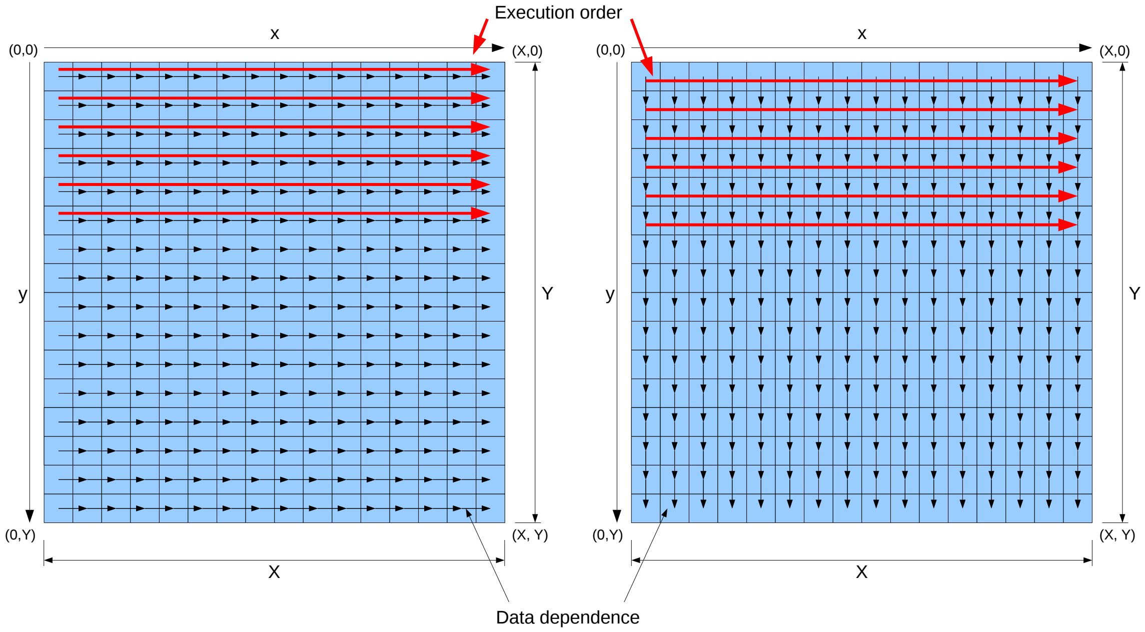
```
for x
  sum[x] = 0.0;
for y
  for x
    sum[x] += input[y*X + x];
for x
  output[x] = sum[x];
```

Here it is flattened into a single, predicated loop nest, ready to be transcribed into a dataflow design:

```
int count = 0;

for (int y=0; y<Y; y += 1) {
  for (int x=0; x<X; x += 1) {
    sum[x] = ((y == 0 ) ? 0.0 : sum[x]);
    sum[x] += input[count]
    count += 1;
    if (y == Y-1) {
      output[x] = sum[x];
    }
  }
}
```

When doing a column sum in this manner, each iteration of the inner loop is not dependent on the previous iteration of the inner loop (x), as it is in a row sum, but is instead dependent on the previous iteration of the outer loop (y). This increased *dependence distance* helps us because we can push a different sum with each input into the adder pipeline every Kernel tick, as we have plenty of time for the result to come back round from the adder output (as long as the pipeline depth of our adder is smaller than the height of the input array Y).

Figure 7 shows how the dependence distance differs. The black arrows show data dependency and the red arrows show the execution order, or the order in which data is streamed into the Kernel. With row-wise summation, each iteration depends on the previous one (dependence distance = 1). For summation column-wise, each iteration depends on a value from X iterations earlier, so the dependence distance is X.



(a) row-sum

(b) column-sum

Figure 7: Execution order and data dependence for row-sum (a) and column-sum (b)

15 Input Transposition

Why this helps us with our original row-sum problem may not be immediately obvious, but if we can *transpose* the input data such that we receive it column-wise in our Kernel, then we can do a column sum on the data in the Kernel to give the row-sum result in the output. This means that we can then get throughput of one add per tick from our Kernel, which is a dramatic improvement on the previous multi-tick implementation. *Figure 8* shows the actual execution order and data dependency that we have achieved using this transposition.

Note that in the code and description for this example, the Kernel performs a column sum with an X and Y swapped over from the X and Y of the original row-sum problem.

The Kernel design is shown in *Listing 9*. The Kernel graph for this design is shown in *Figure 9*. We now have a continuous input, which is going to consume a new value every tick:

```
        DFEVar input = io.input(”input”, scalarType);
```

We have a counter chain that counts the rows (y) using the outer counter and the columns (x) using the inner counter:

```
        CounterChain chain = control.count.makeCounterChain();
        DFEVar y = chain.addCounter(Y, 1);
        chain.addCounter(X, 1); // x
```

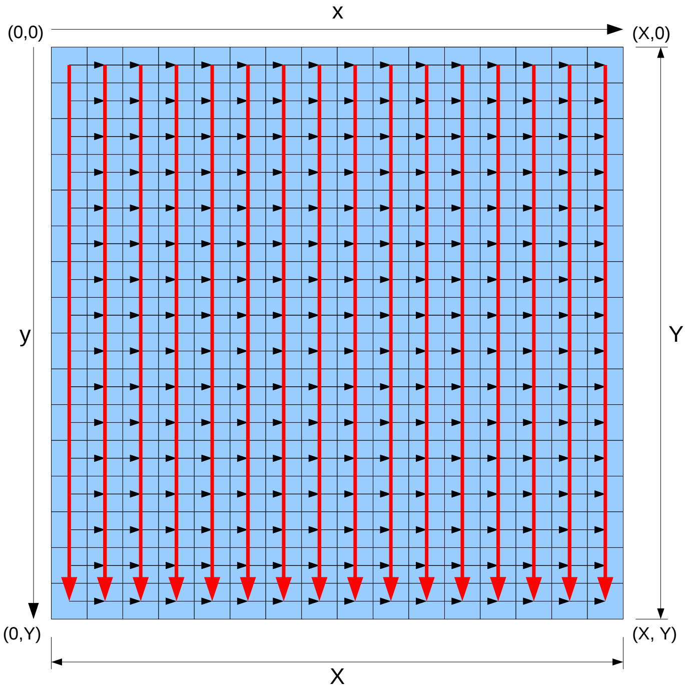


Figure 8: Execution order and data dependence for transposed input solution to row-sum problem

The multiplexer selecting between the sum and 0 is now controlled by the row counter `y`:

```
        DFEVar carriedSum = scalarType.newInstance(this); // sourceless, connected later
        DFEVar sum = y === 0 ? 0.0 : carriedSum;
```

The sum is exactly the same as before:

```
        DFEVar newSum = input + sum;
```

The backward edge is connected with an offset of `-X`, the *width* of the array:

```
        carriedSum <== stream.offset(newSum, -X);
```

Finally, the output is now controlled by the counter `y` reaching the end of a column, so we do an add every tick and get a result at the end of every column:

```
        io.output("output", newSum, scalarType, y === (Y - 1));
```

We no longer need to output from the offset as we did in the cyclic implementation previously, because the immediate output from the adder at the end of each column is correct.

The CPU code for this Kernel (shown in *Listing 10* and *Listing 11*) needs to transpose the input

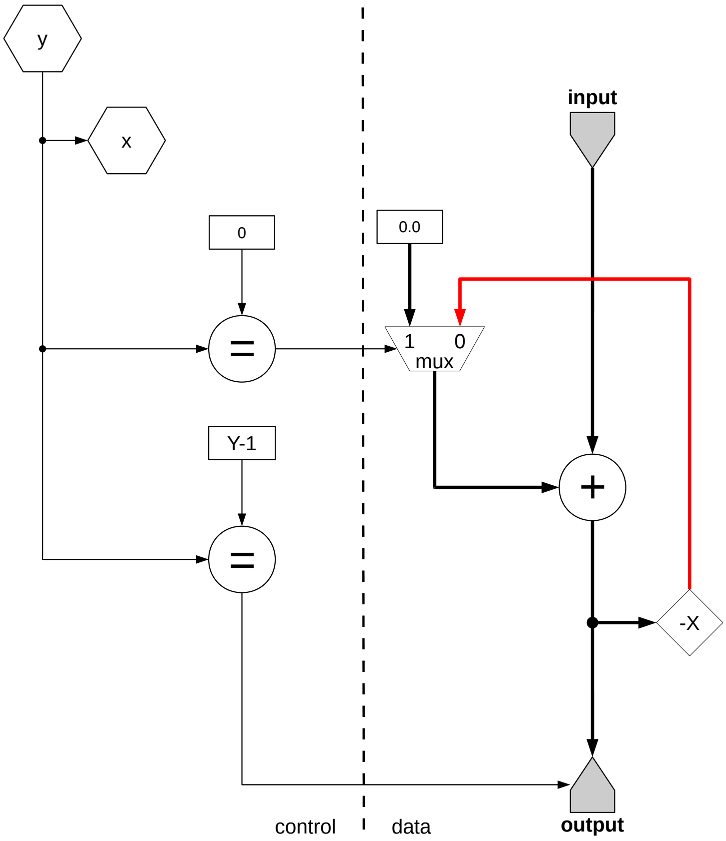


Figure 9: Kernel graph for loop-tiled row-sum

Listing 9: Class for the column-wise summation Kernel (ColSumKernel.maxj).

```
/**
 * Document: Acceleration Tutorial - Loops and Pipelining (maxcompiler-loops.pdf)
 * Example: 8      Name: Column sum
 * MaxFile name: ColSum
 * Summary:
 *      Kernel design that implements a column wise summation.
 */
package colsum;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.CounterChain;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class ColSumKernel extends Kernel {

    final DFEType scalarType = dfeFloat(8,24);

    ColSumKernel(KernelParameters parameters, int X, int Y) {
        super(parameters);

        // Input
        DFEVar input = io.input(”input”, scalarType);

        // Set up counters for 2D loop
        CounterChain chain = control.count.makeCounterChain();
        DFEVar y = chain.addCounter(Y, 1);
        chain.addCounter(X, 1); // x

        // At the head of the loop, we select whether to take the initial value,
        // or the value that is being carried around the data flow cycle
        DFEVar carriedSum = scalarType.newInstance(this); // sourceless, connected later
        DFEVar sum = y === 0 ? 0.0 : carriedSum;

        // The loop body itself
        DFEVar newSum = input + sum;

        // At the foot of the loop, we add the backward edge
        carriedSum <== stream.offset(newSum, -X);

        // We have a controlled output to deliver the sum at the end of each column
        io.output(”output”, newSum, scalarType, y === (Y - 1));
    }
}
```

stream:

```
    // Transpose the inputs
    for (int i = 0; i < X; ++i)
        for (int j = 0; j < Y; ++j)
            inputTransposed[j + Y * i] = input[j * X + i];
```

This transposition works with an array that does not have equal X and Y dimensions, as long as the number of rows is larger than the pipeline depth of the adder.

Note that the dimensions for `X` and `Y` passed to the Kernel in the Manager need to be swapped to match the transposed inputs:

```
        Kernel kernel = new ColSumKernel(manager.makeKernelParameters(), Y, X);
```

Listing 10: CPU code for the column-wise summation Kernel (ColSumCpuCode.c part 1).

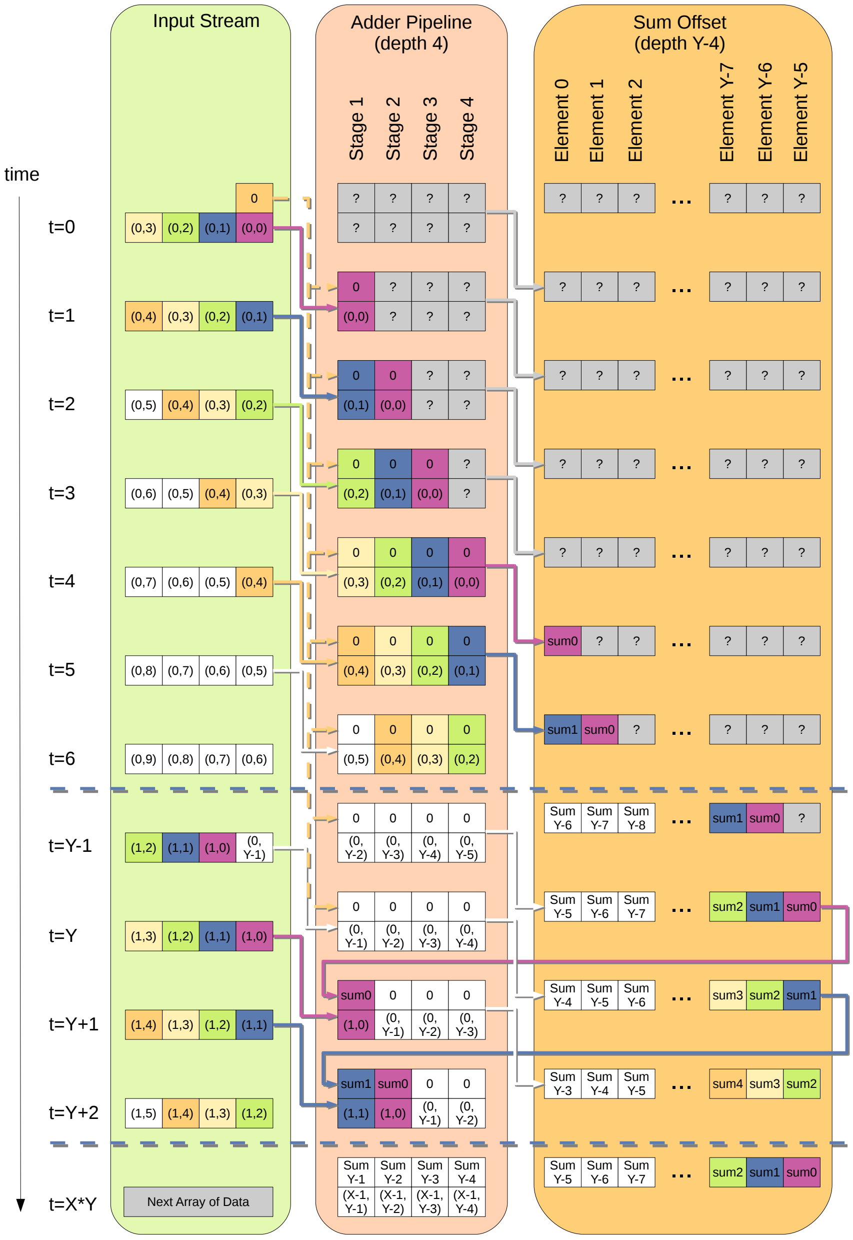
```
/**
 * Document: Acceleration Tutorial - Loops and Pipelining (maxcompiler-loops.pdf)
 * Example: 8      Name: Column sum
 * MaxFile name: ColSum
 * Summary:
 *      CPU code for the column-wise summation design. Ensures inputs are
 *      transposed to match the order they are consumed in by the kernel.
 */
#include <stdlib.h>
#include <stdint.h>

#include <MaxSLiCInterface.h>
#include ”Maxfiles.h”

const int X = ColSum_X; // image row length (number of columns)
const int Y = ColSum_Y; // image column length (number of rows)

void generateInputData( float *inputTransposed)
{
    const int size = X * Y;
    int sizeBytes = size * sizeof(float);
    float *input = malloc(sizeBytes);

    for (int y = 0; y < Y; ++y)
        for (int x = 0; x < X; ++x)
            input[y * X + x] = (y + x);

    // Transpose the inputs
    for (int i = 0; i < X; ++i)
        for (int j = 0; j < Y; ++j)
            inputTransposed[j + Y * i] = input[j * X + i];
    free(input);
}

void ColSumCPU(float *input, float *output)
{
    for (int y = 0; y < Y; ++y) {
        output[y] = 0.0;
        for (int x = 0; x < X; ++x) {
            output[y] += input[x * Y + y];
        }
    }
}

int check(float *output, float *expected)
{
    int status = 0;
    for (int i = 0; i < Y; i++) {
        if (output[i] != expected[i]) {
            fprintf(stderr, ”[%d] error, output: %f != expected: %f\n”,
                i, output[i], expected[i]);
            status = 1;
        }
    }
    return status;
}
```

In the case of our example, we don't need to transpose the output as the column results come out in the correct order: in some applications, it may be necessary to transpose the output as well as the input.

Listing 11: CPU code for the column-wise summation Kernel (ColSumCpuCode.c part 2).

```
int main()
{
    const int size = X * Y;
    int dataSizeBytes = size * sizeof(float);
    float *input = malloc(dataSizeBytes);
    float *output = malloc(dataSizeBytes);
    float *expected = malloc(dataSizeBytes);

    generateInputData(input);

    printf (”Running DFE.\n”);
    ColSum(size, input, output);

    ColSumCPU(input, expected);

    int status = check(output, expected);
    if (status)
        printf (”Test failed.\n”);
    else
        printf (”Test passed OK!\n”);

    return status;
}
```

The Kernel need only be run for `X * Y` ticks as we now have a throughput of one add per tick. *Figure 10* shows the now fully-utilized adder pipeline (again with a pipeline depth of 4 to make the diagram simpler) and the offset buffer which loops back to the input of the adder. Note that the length of the offset buffer is 4 less than that the offset `X` set in the code as 4 elements are balanced with the pipeline depth of the adder.



The pipeline is empty.

The adder starts computation of the new sum for row 0.

The adder starts computation of the new sum for row 1.

The adder starts computation of the new sum for row 2.

The adder starts computation of the new sum for row 3.

The adder starts computation of the new sum for row 4. The first addition for the sum of row 0 is inserted into the offset buffer.

The adder starts computation of the new sum for row Y-1 (the last row).

The sum for row 0 has passed through the offset buffer. The adder starts computation of the new sum for row 0.

The first complete row sum for row 0 is output.

Figure 10: Fully-used pipeline in the transposed column implementation of the row-sum

16 Loop Tiling

Loop tiling is a general technique for transforming a loop nest to operate on blocks of the iteration space at a time. We will use loop tiling to gain control over the dependence distance, so that we can match it precisely to the pipeline depth - and so avoid unnecessary buffering. This use of tiling is sometimes referred to as **C-Slow retiming** in the hardware community[1].

In *section 15*, we transposed the input for our complete array. For small arrays, this is feasible, but for larger arrays this creates a large amount of buffering logic (as we can see from *Figure 10*), which can make very inefficient use of valuable resources in the DFE or potentially make the design unfeasibly large. We will use loop tiling to split the transposition into smaller blocks which will maintain our throughput of one add per tick, but reduce our logic utilization.

16.1 Overview of Loop Tiling

The basic idea is of loop tiling is:

- The cycle in our data flow graph has at least C pipeline stages in it (13, due mainly to the floating-point adder, in our case)
- The value being calculated at each stage can depend on the output of the *same stage* from C ticks earlier
- Each of these stages is an opportunity to do one summation
- So, we need C different, independent summations to fill each pipeline stage in the tick in the data-flow graph
- We reorder the inputs to arrive in transposed blocks of C rows
- The resulting implementation will now have a throughput of one calculation per tick

We present the idea here in terms of a loop nest transformation. We are going to split the loop nest into blocks of C iterations, then interchange the loops to create the dependence distance we need.

Consider the original row-summation loop again:

```
int count = 0;
for (int y=0; ; y += 1) {
  sum[y] = 0.0;
  for (int x=0; x<X; x += 1) {
    sum[y] = sum[y] + input[count];
    count += 1;
  }
  output[y] = sum[y];
}
```

[1] A useful overview can be found in http://brass.cs.berkeley.edu/documents/Cslow_Retiming_Virtex.pdf.

First we **strip-mine** the outermost loop, which means splitting it into chunks of size C:

```
int count = 0;
for (int yy=0; ; yy += C) {
  for (int y=yy; y < yy+C; y += 1) {
    sum[y] = 0.0;
    for (int x=0; x<X; x += 1) {
      sum[y] = sum[y] + input[count]
      count += 1;
    }
    output[y] = sum[y];
  }
}
```

Now we want to do a loop interchange, but the y loop has three statements, of which one is a loop. We can fix this by shifting these extra statements into the innermost loop, predicating them as we did in *section 8* so that they implement the same behavior:

```
int count = 0;
for (int yy=0; ; yy += C) {
  for (int y=yy; y < yy+C; y += 1) {
    for (int x=0; x<X; x += 1) {
      if (x == 0) sum[y] = 0.0;
      sum[y] = sum[y] + input[count]
      count += 1;
      if (x == X-1) output[y] = sum[y];
    }
  }
}
```

Now we can interchange the inner two loops:

```
int count = 0;
for (int yy=0; ; yy += C) {
  for (int x=0; x<X; x += 1) {
    for (int y=yy; y < yy+C; y += 1) {
      if (x == 0) sum[y] = 0.0;
      sum[y] = sum[y] + input[count]
      count += 1;
      if (x == X-1) output[y] = sum[y];
    }
  }
}
```

Figure 11 shows the execution order and data dependency of this method, with the input broken up into groups of rows of height C. In this case, C is 4 to make the diagram clearer.

With this execution order, input is consumed in a sequence of $C \times M$ chunks, each of which is processed column-wise in turn.

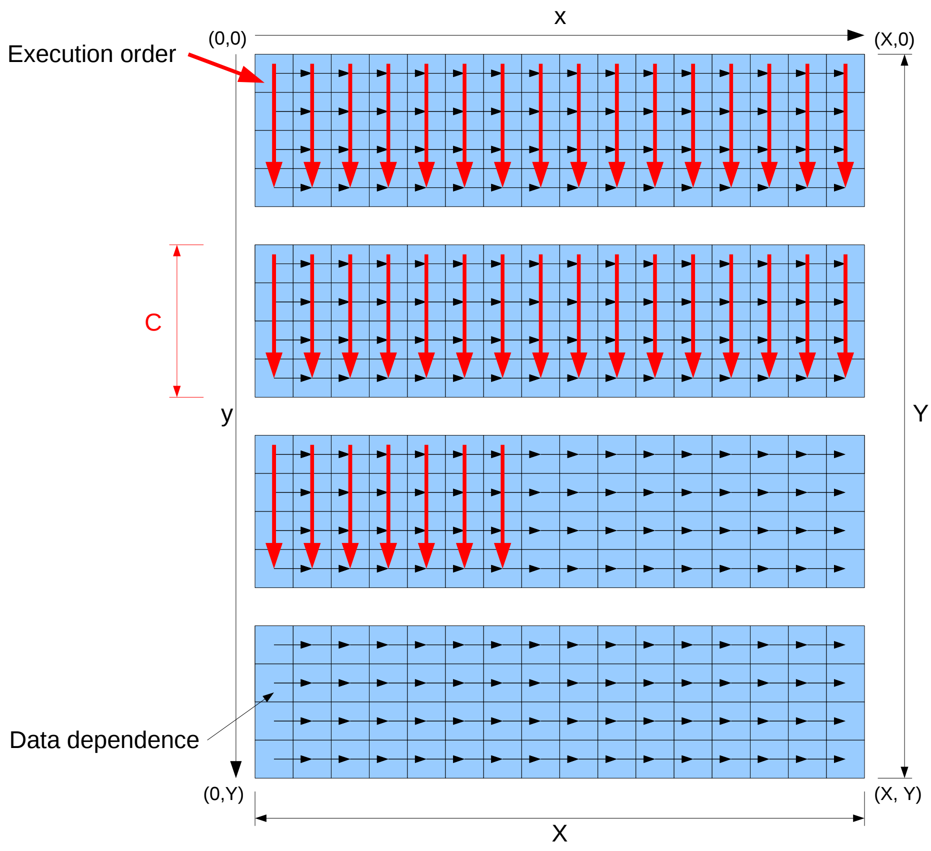


Figure 11: Execution order and data dependence for loop-tiled row-sum, with C of 4

16.2 Loop-Tiled Implementation of the Row-Sum

Example 9 shows a loop-tiled implementation of the row-sum problem we saw previously. The Kernel code for this implementation is shown in *Listing 12* and the corresponding Kernel graph in *Figure 12*.

We have a continuous input as before:

```
        DFEVar input = io.input(”input”, scalarType);
```

`C` is a compile-time Java constant defined as a divisor of the X dimension in the Manager:

```
    RowSumLoopTileKernel(KernelParameters parameters, int X, int C) {
        super(parameters);
```

The counter chain is set up with the `y` and `x` dimensions reversed. The inner `yy` counter counts up

Listing 12: Class for the row-sum kernel (RowSumLoopTileKernel.maxj).

```
/**
 * Document: Acceleration Tutorial - Loops and Pipelining (maxcompiler-loops.pdf)
 * Example: 9       Name: Row-sum loop tile
 * MaxFile name: RowSumLoopTile
 * Summary:
 *       Kernel design for a loop- tiled implementation of the row sum problem
 *       using auto loop offsets and distance measurement.
 */

package rowsumlooptile;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.CounterChain;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class RowSumLoopTileKernel extends Kernel {

    final DFEType scalarType = dfeFloat(8, 24);

    RowSumLoopTileKernel(KernelParameters parameters, int X, int C) {
        super(parameters);

        // Input
        DFEVar input = io.input("input", scalarType);

        CounterChain chain = control.count.makeCounterChain();
        DFEVar x = chain.addCounter(X, 1);
        // Set up counter for innermost, y loop, except we count 0..C
        // instead of yy..yy+C
        chain.addCounter(C, 1); // yy

        // At the head of the loop, we select whether to take the initial value,
        // or the value that is being carried around the data flow cycle
        DFEVar CarriedSum = scalarType.newInstance(this); // sourceless, connected later
        DFEVar sum = x === 0 ? 0.0 : CarriedSum;

        // The loop body itself
        DFEVar NewSum = input + sum;

        // At the foot of the loop, we add the backward edge
        CarriedSum <== stream.offset(NewSum, -C); //

        // We have a controlled output to deliver the sum at the end of each row
        io.output("output", NewSum, scalarType, x === (X - 1));
    }
}
```

to C:

```
        CounterChain chain = control.count.makeCounterChain();
        DFEVar x = chain.addCounter(X, 1);
        // Set up counter for innermost, y loop, except we count 0..C
        // instead of yy..yy+C
        chain.addCounter(C, 1); // yy
```

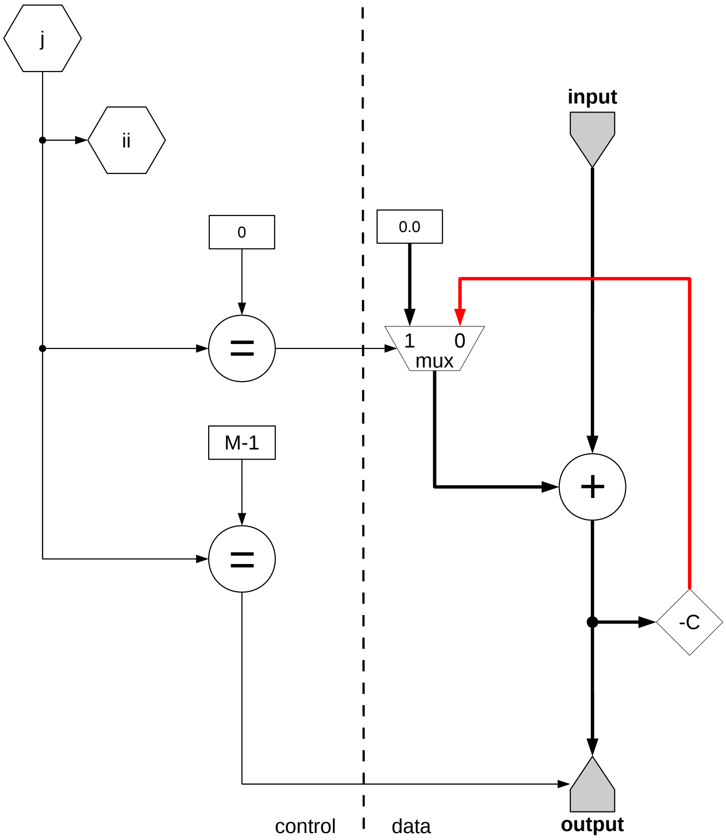


Figure 12: Kernel graph for loop-tiled row-sum

The multiplexer and adder are implemented in exactly the same way as before:

```
DFEVar CarriedSum = scalarType.newInstance(this); // sourceless, connected later
DFEVar sum = x === 0 ? 0.0 : CarriedSum;

// The loop body itself
DFEVar NewSum = input + sum;
```

The backward edge is now offset by the `C`:

```
CarriedSum <== stream.offset(NewSum, -C); //
```

The offset expression is defined as the sum of an AutoLoop offset and a variable stream offset parameter:

```
CarriedSum <== stream.offset(NewSum, -C); //
```

Finally, the sum for each row is output when `x` reaches `X - 1`:

```
io.output("output", NewSum, scalarType, x === (X - 1));
```

One complicating factor is that the loop-tiled version consumes its input in a different order - so the

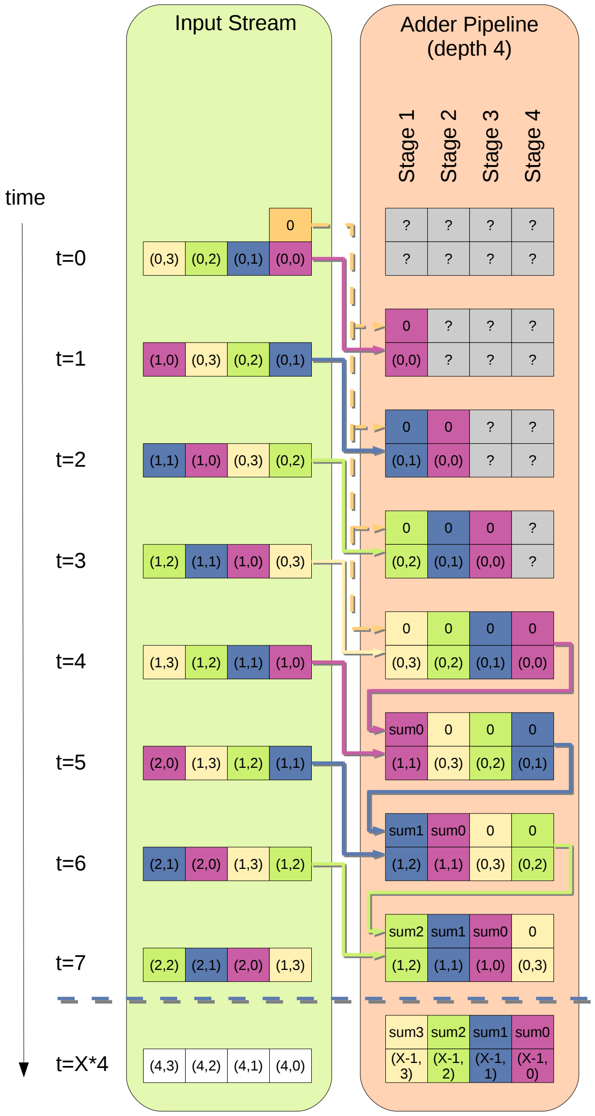
CPU code (*Listing 13* and *Listing 14*) applies a corresponding reordering to the input sequence:

```
// Transpose the inputs
int count = 0;
for (int yy = 0; yy < Y; yy += C) {
    for (int x = 0; x < X; ++x) {
        for (int y = yy; y < yy + C; ++y) {
            inputTransposed[count] = input[y * X + x];
            count++;
        }
    }
}
```

The C factor is passed to the CPU code using a `.max` file constant:

```
const int C = RowSumLoopTile_CFactor; // C Factor used in the DFE
```

Figure 13 shows a pipeline depth of 4 for clarity, with a value of C also of 4, so that there is no buffering required: the output of the adder loops straight back to the input.



The pipeline is empty.

The adder starts computation of the new sum for row 0.

The adder starts computation of the new sum for row 1.

The adder starts computation of the new sum for row 2.

The adder starts computation of the new sum for row 3.

The output of the first addition for row 0 loops straight back to the input.
The adder starts computation of the new sum for row 0.

The output of the first addition for row 1 loops straight back to the input.
The adder starts computation of the new sum for row 1.

The output of the first addition for row 2 loops straight back to the input.
The adder starts computation of the new sum for row 2.

The first complete row sum for row 0 is output.

Figure 13: Fully-used pipeline in the loop-tiled implementation of the row-sum

Listing 13: CPU Code for the loop-tiled row-sum (RowSumLoopTileCpuCode.c - part 1).

```
/**
 * Document: Acceleration Tutorial - Loops and Pipelining (maxcompiler-loops.pdf)
 * Example: 9       Name: Row-sum loop tile
 * MaxFile name: RowSumLoopTile
 * Summary:
 *       CPU code for a loop-tiled implementation of the row sum problem.
 *       Determines the variable offset by carrying out the necessary arithmetic
 *       on integer values derived from the AutoLoop offset. Ensures inputs are
 *       transposed to match the order they are consumed in by the kernel.
 */

#include <stdlib.h>
#include <stdint.h>

#include ”Maxfiles.h”
#include <MaxSLiCInterface.h>

const int X = RowSumLoopTile_X; // image row length (number of columns)
const int Y = RowSumLoopTile_Y; // image column height (number of rows)
const int C = RowSumLoopTile_CFactor; // C Factor used in the DFE

void generateInputData(float *inputTransposed)
{
    const int size = X * Y;
    int sizeBytes = size * sizeof(float);
    float *input = malloc(sizeBytes);

    for (int y = 0; y < Y; ++y)
        for (int x = 0; x < X; ++x)
            input[y * X + x] = (y + x);

    // Transpose the inputs
    int count = 0;
    for (int yy = 0; yy < Y; yy += C) {
        for (int x = 0; x < X; ++x) {
            for (int y = yy; y < yy + C; ++y) {
                inputTransposed[count] = input[y * X + x];
                count++;
            }
        }
    }

    free(input);
}

void RowSumLoopTileCPU(float *input, float *output)
{
    int count = 0;
    for (int yy=0; count < X*Y; yy += C) {
      for (int x=0; x<X; x += 1) {
        for (int y=yy; y < yy+C; y += 1) {
          if (x == 0) {
              output[y] = 0.0;
          }
          output[y] = output[y] + input[count];
          count++;
        }
      }
    }
}

int check(float *output, float *expected)
{
    int status = 0;
    for (int i = 0; i < Y; i++) {
        if (output[i] != expected[i]) {
            fprintf (stderr, ”[%d] error, output: %f != expected: %f\n”,
                i, output[i], expected[i]);
            status = 1;
        }
    }
    return status;
}
```

Listing 14: CPU Code for the loop-tiled row-sum (RowSumLoopTileCpuCode.c - part 2).

```
int main()
{
    const int size = X * Y;
    int sizeBytes = size * sizeof(float);
    float *input = malloc(sizeBytes);
    float *output = malloc(sizeBytes);
    float *expected = malloc(sizeBytes);

    generateInputData(input);

    printf("Running DFE.\n");
    RowSumLoopTile(size, input, output);

    RowSumLoopTileCPU(input, expected);

    int status = check(output, expected);
    if (status)
        printf("Test failed.\n");
    else
        printf("Test passed OK!\n");

    return status;
}
```

17 Reducing Latency

As we have seen, MaxCompiler is geared towards producing highly-pipelined streaming designs, which offer the highest performance. Deeply-pipelined designs do, in general, use significant storage (flip-flop) resources, however, to reach higher clock speeds. In some circumstances, reducing the latency of operations can help reduce the resources required to implement a design or increase its throughput.

17.1 Pipelining Factor

The latency of operations in a Kernel graph can be reduced by using optimization directives. These are available via static methods in the `optimization` field:

```
void optimization.pushPipeliningFactor(double pipelining)
double optimization.peekPipeliningFactor()
void optimization.popPipeliningFactor()
```

MaxCompiler attempts to reduce the latency of operations between calls of `pushPipeliningFactor` and `popPipeliningFactor`. The `pipelining` argument for `pushPipeliningFactor` is a `double` value between `0.0` and `1.0`, where `1.0` suggests no reduction in latency and `0.0` suggests the largest possible reduction in latency (potentially to `0` ticks). MaxCompiler does not guarantee that either any reduction is performed or that an absolute value of `pipelining` produces a particular latency: `pushPipeliningFactor` is a just suggestion to the compiler.

17.2 Fixed Point versus Floating Point

Consider again our original attempt at implementing the row-sum problem in *section 10*, where we tried to use an offset of `-1`. If we change the incoming data type to fixed-point, then an add takes a single tick (again, we can determine this from the MaxCompiler graph output). In order to reset the accumulator for each row with a `0`, we need to multiplex in a 0 with the output of the adder, which gives a combined latency of 2 ticks for the multiplex-accumulate with fixed-point data. We can use `pushPipeliningFactor(0)` to reduce the latency of the multiplexer to 0 ticks:

```
        optimization.pushPipeliningFactor(0);
        DFEVar sum = x === 0 ? 0.0 : carriedSum;
        optimization.popPipeliningFactor();
```

This gives us a single-tick accumulator and restores our throughput of 1 addition per tick, without having to apply loop-unrolling or loop-tiling.

In this case, the logic required for a two-input multiplexer is minimal, so there is little impact on clock speed. For more significant latency reductions, for example on wider multiplexers or arithmetic operations, reducing the latency results in a reduced clock speed, but with the benefit of reduced resource utilization.

18 Conclusion

This tutorial has focused on optimizing a pipelined loop using loop unrolling, input transposition and loop tiling. These are key methods for exploiting application parallelism in MaxCompiler designs. The next stages in improving performance are replicating a single pipeline and optimizing data types.