Petlje

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:

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

1 Uvod

Put od softverske implementacije do dataflow Kernel uključuje primenu petlji. Prikazaćemo više načina za implementaciju petlji unutar protočnih arhitektura.

Počećemo sa jednostavnim petljama da uočimo kako se mogu implementirati (unutar Kernela). Potom ćemo prikazati malo složenije primere koje uključuju kondicionalne petlje. Pokrićemo prikaz tehnika koje se mogu primeniti u aktulenim Kernel implementacijama kao što su: loop unrolling, loop predication, input transposition, loop tiling.

2 The Implicit Outer Loop

Uočite petlju:

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

Kada je implementiramo kao dataflow Kernel, ulazni niz A je stream-ovan u DFE (na primer od CPU memorije do DFE). Slično, izlazni niz B će biti na izlazu stream-ovan od dataflow Kernel do CPU.

Listing 1 **prikazuje Kernel implementaciju ovog trivijalnog primera. Posmatranjem bloka koda u Kernel-u, vidimo da nema petlje:**

```
20      // Input
21      DFEVar input = io.input("input", dfeUInt(32));
22
23      DFEVar result = input + 1;
24
25      // Output
26      io.output("output", result, dfeUInt(32));
```

Petlja u polaznom primeru je zapravo mehanizam za adresiranje niza. U dataflow implementaciji, podatak se stream-uje u Kernel iz Manager-a, tako da ne postoji potreba za petljom unutar Kernel-a: jer petlja implicitno egzistira unutar toka (stream-a).

Manager kontroliše streamovanje podataka od poečetka (npr. podaci koji dolaze direktno iz CPU, ili iz LMem sa DFE ili preko MaxRing sa druge DFE). U ovom primeru, pristup podacima pokreće CPU code tako što se kopira ulazni niz A u DFE, a potom kopira nazad niz B .

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

```
1  /**
2   * Document: Acceleration Tutorial - Loops and Pipelining (maxcompiler-loops.pdf)
3   * Example: 1       Name: Increment
4   * MaxFile name: Increment
5   * Summary:
6   *      Kernel design that increments the input stream and sends the
7   *      result to the output stream.
8   */
9  package increment;
10
11 import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
12 import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
13 import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
14
15 class IncrementKernel extends Kernel {
16
17     IncrementKernel(KernelParameters parameters) {
18         super(parameters);
19
20         // Input
21         DFEVar input = io.input("input", dfeUInt(32));
22
23         DFEVar result = input + 1;
24
25         // Output
26         io.output("output", result, dfeUInt(32));
27     }
28 }
```

3 Petlja - brojač

Pogledajmo sličan primer petlje i brojačku promenljivu `count` :

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

Listing 2 je primer DFE implementacije ovog primera. I dalje unutar Kernela se ne pojavljuje petlja, ali je dodat brojač tj.counter radi evidencije broja iteracije za svaki ulazni podatak

.

```
19         DFEVar input = io.input("input", dfeUInt(32));
20
21         DFEVar counter = control.count.simpleCounter(32);
22
23         DFEVar result = input + counter;
24
25         // Output
26         io.output("output", result, dfeUInt(32));
```

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

```
1  /**
2   * Document: Acceleration Tutorial - Loops and Pipelining (maxcompiler-loops.pdf)
3   * Example: 2       Name: Simple counter
4   * MaxFile name: SimpleCounterLoop
5   * Summary:
6   *      A simple kernel that adds the value from a counter to a stream.
7   */
8  package simplecounterloop;
9
10 import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
11 import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
12 import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
13
14 class SimpleCounterLoopKernel extends Kernel {
15     SimpleCounterLoopKernel(KernelParameters params) {
16         super(params);
17
18         // Input
19         DFEVar input = io.input("input", dfeUInt(32));
20
21         DFEVar counter = control.count.simpleCounter(32);
22
23         DFEVar result = input + counter;
24
25         // Output
26         io.output("output", result, dfeUInt(32));
27     }
28 }
```

4 Ugnježđena petlja

Uočite sledeću ugnježđenu petlju:

```
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 prikazuje Kernel koji implementira ugnježđenu petlji.

Kod Kernela uključuje lanac brojača da bi se obezbedilo postojanje para indeksa za svaki ulazni podatak:

```
25         // Set up counters for 2D loop
26         CounterChain chain = control.count.makeCounterChain();
27         DFEVar y = chain.addCounter(Y, 1).cast(dfeUInt(32));
28         DFEVar x = chain.addCounter(X, 1).cast(dfeUInt(32));
29
30         DFEVar result = input + (y * 100) + x;
```

Telo petlje se izvršava za svaki ulaz. Promenljiva count u primeru ugnježđene petlje je implicitno prisutna u listingu 3, tj. implicitno je prisutna u stream-u.

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

```
1  /**
2   * Document: Acceleration Tutorial - Loops and Pipelining (maxcompiler-loops.pdf)
3   * Example: 3       Name: Simple two-dimensional counter
4   * MaxFile name: Simple2dCounter
5   * Summary:
6   *      Kernel design that implements a nested loop computation using a counter
7   *      chain to provide a pair of indices for each data value presented on the
8   *      input.
9   */
10 package simple2dcounter;
11
12 import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
13 import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
14 import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.CounterChain;
15 import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
16
17 class Simple2dCounterKernel extends Kernel {
18
19     Simple2dCounterKernel(KernelParameters params, int X, int Y) {
20         super(params);
21
22         // Input
23         DFEVar input = io.input("input", dfeUInt(32));
24
25         // Set up counters for 2D loop
26         CounterChain chain = control.count.makeCounterChain();
27         DFEVar y = chain.addCounter(Y, 1).cast(dfeUInt(32));
28         DFEVar x = chain.addCounter(X, 1).cast(dfeUInt(32));
29
30         DFEVar result = input + (y * 100) + x;
31
32         // Output
33         io.output("output", result, dfeUInt(32));
34     }
35 }
```

5 Zavisnost podataka

Uočite petlju koja implementira Newton-Raphson recipročnu aproksimaciju za 0.5 < d < 1.0 (nula funkcije f(x)=1/x-d), gde četiri iteracije su dovoljne za 32-bit tačnost:

```
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).

```
1  /**
2   * Document: Acceleration Tutorial - Loops and Pipelining (maxcompiler-loops.pdf)
3   * Example: 4       Name: Reciprocal
4   * MaxFile name: Reciprocal
5   * Summary:
6   *      Kernel design that implements a reciprocal using the Newton-Raphson
7   *      method.
8   */
9  package reciprocal;
10
11 import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
12 import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
13 import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
14
15 class ReciprocalKernel extends Kernel {
16
17     ReciprocalKernel(KernelParameters params) {
18         super(params);
19
20         // Input
21         DFEVar d = io.input("d", dfeFloat(8, 24));
22
23         DFEVar v = 2.9142 - 2.0*d;
24
25         for (int iteration = 0; iteration < 4; iteration += 1) {
26             v = v*(2.0 - d*v);
27         }
28
29         // Output
30         io.output("output", v, dfeFloat(8, 24));
31     }
32 }
```

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*:

```
25         for (int iteration = 0; iteration < 4; iteration += 1) {
26             v = v*(2.0 - d*v);
27         }
```

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 de-

liver 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.

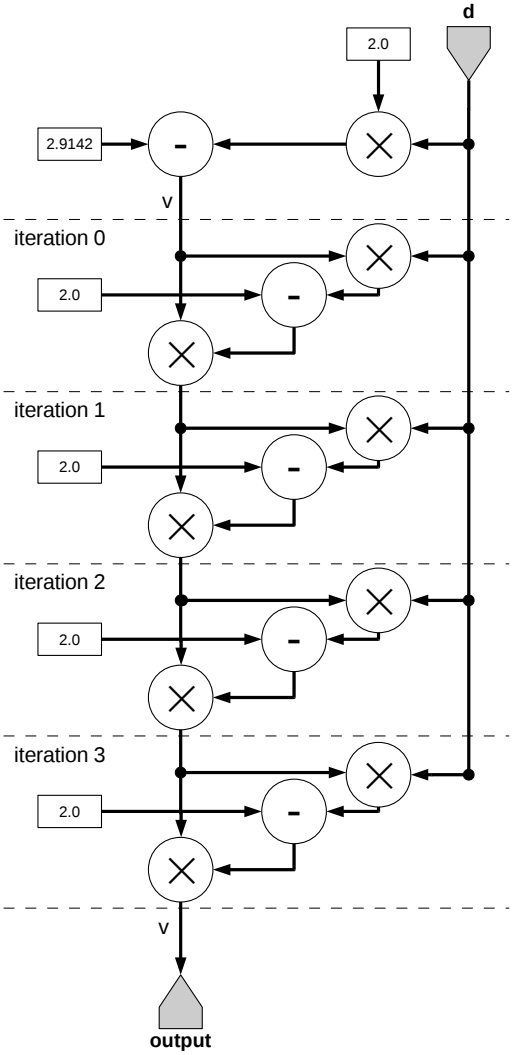

Figure 1: Kernel graph for an unrolled Newton-Raphson reciprocal approximation

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.

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:

```
34      optimization.pushPipeliningFactor(0);
35      DFEVar sum = x === 0 ? 0.0 : carriedSum;
36      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.