Summer Research Reports

Vincent Zhao
vincentzhaorz@gmail.com

September 3, 2015

Abstract

This is the 7th week of my summer research, and this report will contain mainly what I've done in the past 2 weeks, including MaxJ SpMV inplementation, Ruby multipumping implementation, and some bug fix in Ruby compiler.

Contents

1 Week7: BCSR and Multi-pumped Implementation

1.1 What is BCSR?

BCSR is a sparse matrix format which is based on well-known CSR format. It's basic element is little block of non-zeros, just like cutting CSR to little blocks. Each block has a shape of $r \times c$, where r and c's value will not be so large.

The BCSR format contains 2 main procedures, one is converting, which will transform a traditional COO format to BCSR format storage. The other one is computing, which will do computation on BCSR format.

1.1.1 Convert

The convert procedure will act like this, for example here's a sparse matrix A:

$$A = \begin{bmatrix} a & 0 & b & c \\ d & e & f & 0 \\ 0 & 0 & 0 & g \\ 0 & h & i & 0 \end{bmatrix}$$

If it's a CSR format, the final result would be:

$$A_{csr} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & & \\ h & i & \end{bmatrix}$$

And for a BCSR format, the result would be:

$$A_{bcsr} = \begin{bmatrix} a & b \\ d & e \end{bmatrix}, \begin{bmatrix} c & 0 \\ f & 0 \end{bmatrix}, \begin{bmatrix} g & 0 \\ h & i \end{bmatrix}$$

1.1.2 Compute

1.2 Why choose BCSR on FPGA?

1.3 BCSR Implementation

1.4 BCSR Performance

Let's check out the maximal performance we could get if this design is IO-bound. If the PCIe bandwidth is 2GB/s, and we'll input 2 32-bits data, one for index, the other one for value, for each non zero. So we could process data per second:

$$\text{Non zero per second} = \frac{2(GB/s)}{2 \times 4(Byte)} = 0.25(G/s)$$

And for each non zero, we will have 2 floating point operations(adder and multiplier), so the maximal GFlops we could get is:

$$\text{GFlops} = 0.25(G/s) \times 2(Flop) = 0.5$$

1.5 Multi-pumped BCSR design

Referring to the BCSR format, we will calculate multiple multiplication in one single tick, which is a brillliant feature to be multi-pumped. But due to the constraints of MaxCompiler, we need to have multiple kernel blocks now, as we could only assign one universal clock rate to each kernel. We could divide this multi-pumped BCSR design into 3 parts:

Gather For each pipe, we need to gather the input vector and value vector. Input vector means the values in the vector parameter, read from ROM.

Multiply We send those input and value vectors to this multiply part, and this part runs at 2x clock rate, which is the computation core of multi-pumping design.

Sum Will take result from multi-pumping multipliers and use summation tree to sum them up. It will also take a `start` stream to denote whether the current calculated block is the start of each group.

2 Week 7: Ruby serial primitives

This part of work is about how to compile Ruby serial primitives to MaxJ code, mainly `pdsr` and `sdpr`.

2.1 `showGate` and Function Device

`showGate` is the core function to generate the MaxJ expression of a Ruby circuit. For instance, `add` primitive will be transformed to "+" by using `showGate`. But things will be different for `pdsr` and `sdpr`, as `showGate` currently could only support infix, eq and some other basic primitives, there's nothing like "function" has been supported, which will be quite useful to definie `pdsr` and `sdpr`.

Briefly, my approach is:

1. Add a condition filter `isFunc` in `showGate`, to check whether the current device is a **function device**, which should be implemented like `func(t1, t2)` in MaxJ.
2. Append the builtin function definition backward in the kernel class definition.

2.2 `pdsr` and `sdpr`

How to define these 2 primitives in MaxJ? The simplest way is using a combination of **counter** and **mutiplexer**, which should be `simpleCounter` and `mux` in MaxJ. A simple definition for `pdsr 2` is:

```
private DFEVar pdsr2(DFEVar t0,DFEVar t1) {
  DFEVar counter = control.count.simpleCounter(
                        MathUtils.bitsToAddress(2));
  return control.mux(counter, t0, t1);
}
```

This snippet of code is auto-generated, and suit for different value of n in `pdsr n` expression.

But for `sdpr`, things will be different, as no 2 or more number of output ports device will be supported in `printmax`, I think maybe it's better to optimize this by using `DFEVector`.

3 Reports Archive

3.1 Report of Week 3 (Aug 3rd, 2015)

Basically I've finished running and testing the real hardware implementation of my previous SpMV design, and improved it by enabling large matrix calculation. Also, I've written a simple multi-pumped design by using MaxJ. This one has been checked about its accuracy and time frequency.

3.1.1 SpMV Implementation

CSRP format The format that I've devised and implemented could be called as CSRP (*CSR format with Padding zeros*). This format enables multiple rows of elements to be calculated at the same time, and when those simultaneous caculated rows don't have the same number of non-zeros, this format will automatically add padding zeros to ensure the result is correct.

The main advantage of this format is its simplicity in implementation, but it will also face these deflects:

1. *Redundant calculation*: For those cases when different rows have quite different number of non-zeros, the total number of calculations will depend on the maximum number of non-zeros among these rows.

2. *Resource duplication*: We need to replicate same kind of resources to enable parallelized calculation.

Implementation The implementation of CSRP format is quite simple. We set 2 input streams, one is for value and the other is for column index of each non-zero element. All these input streams are interleaved and zero-padded. Each of the calculation components has one a cyclic-loop which will sum the result for current row, and one ROM which stores the values of vector. We take Matrix Market Format of matrix as input, and generate interleaved data from the original data.

Optimization The detailed statistics will be updated by next week, but here I'll list several ideas of optimization:

1. *Reduce redundant calculation*: Currently the number of redundant calculation depends on the maximal number of non-zeros of all rows in the matrix. We could reorganize our output data and let number of padding zeros depend on the maximal number of non-zeros of all rows in each simultaneously calculated groups. After that, we could also reorder the matrix rows to get the best placement of rows into different groups, and get a globally optimizated result.

2. *Summation tree*: Current version of SpMV is based on a cyclic loop, which will sum the results for each row. A summation tree implementation should be better.

3. *Reduce resource usage*: We could do this by using dual-port of ROM, and if possible, using multi-pumping.

TODO

1. Finish the fully implemented baseline by Thursday. Fully implemented means, it could take MMF matrix data as input, and has reasonable space of memory.

2. Finish the first kind of optimization and the dual-port version by the end of this week.

3. If possible, try a multi-pumped version. (Only after the first 2 tasks finished)

3.1.2 Multi-Pumping

Simple implementation This simple implementation will do square on 2 input streams. Those input streams will be configured at 1x clock rate, and the inner multiplier will be configured at 2x clock rate. It could be implemented by using `ManagerClock` in MaxJ. The real hardware implementation has also been checked. The result is correct but the frequency is not accurate.

TODO

1. Finish checking this simple implementation. Must get a reasonable clock frequence.

2. Finish a Ruby implementation of multi-pumping by Thursday. Also, generate its MaxJ code.

3.1.3 Ruby Compiler

I haven't checked the compiler last week, but I'll start it as soon as possible. Basically, I should first use ruby to do some basic design, compile it into MaxJ and run it. After that, try to do some loop in Ruby. And test its MaxJ result (Which may not be correct).

TODO

1. *Loops in Ruby*: Test loops in Ruby, including acyclic loops and cyclic loops. Simulate them and check its generated MaxJ code.

3.2 Report of Week 4 (Aug 4th, 2015)

This week I've made progress on all the 3 directions: SpMV in MaxJ, multi-pumping in MaxJ and multi-pumping in Ruby. Here're the details.

3.2.1 SpMV in MaxJ

According to the plan, I've finished the baseline version of CSRp SpMV on time. You could find out the definition of CSRp in the archived week 3 report. This version will generate correct result, but it must be improved in many aspects:

1. **Get runtime correct**: Currently, if we add level of parallelization, the run time will increase, not as what we expected. If the code is correct(maybe not), then something wrong might happen during synthesis and implementation. Check the **clock frequency** of the **built kernel**. This will tell whether there's problem during the implementation: if the clock frequency also not follow our expectation.

   It might be the case that the current using matrix will never benefit from the level of parallelization, so we need to generate matrix based on our current design specification, and discover their performance.

   Todo First check out the frequency of each design with different parallelization level. Also try to organize the code better.Then use randomized benchmark, not only the value, but also the structure to test run time result.

2. **Improved CSRp format**: The CSRp format will perform really bad if the padded zeros occupy most of the computations, which means, we have lots of redundant zeros. Why we have those zeros? As we cannot calculate all the rows simultaneously, only part of the rows will be calculate at the same time. If we call these rows a **block**, then we will calculate the whole matrix block by block. Now, we do not specifically add **splitters** between those blocks. We would know which block we are calculating only by forcing all the blocks has equal width. The way we force this happen is by padding zeros. The number of padding zeros could be enormous if we have a very wide block and most of the blocks are quite narrow. The **redundancy** metrics could be calculated by using:

   $$R_{CSRp}(N) = \max_{0 \le i < N} (block_i.width) \times N \times p - \sum_{i}^{Np} row_i.width$$

   where N is the number of blocks, and p is the level of parallelization. If we have N and p assigned, then the whole number of rows should be Np. The **width** is the number of non-zeros for each row or block. Especially for blocks, their width is the maximal width among all rows in that block.

Obviously, we could improve this. If we assign more information for each clock cycle, like "whether this cycle is in the end of one block", which is the **splitter** that we have mentioned above, then we do not need to do padding on all the blocks to make sure they have same width.

This one could be achieved by adding a new input stream of 0, 1 boolean values. We could verify the improvement in speed and space increasement.

Todo: Finish the first part of SpMV work first. And build this design later. Discover their performance differences. And I think this is one should be the standard design of **baseline**. Previous one has really bad performance.

3. **With multi-pumping**: I've already finished a multi-pumping MaxJ version, which has a simple multiplier inside. And we all know that SpMV is all about add and multiply(basically), so we could use the same framework to optimize this SpMV design.

   Todo: Last one to do, I think I could do this in the next week.

That's all about this week's SpMV work, I'll attach a full chart about this design's performance in next week's report, as I need to verify some current data this week.

3.2.2 Multi-pumping in MaxJ(Updated on Aug 25th 2015)

In this part I'll just illustrate some result I've gathered in my Multi-pumping MaxJ version's experiment.

First of all, we have these versions to implement the simple multiplication functionality:

1. **UnrollLoop**: This one will use 1x clock rate for multiplier, but will use 2 of them.
2. **ResourceShare**: This one will use one single multiplier to do 2 multiplications sequentially, at 1x clock rate.
3. **Multi-pumping**: This one will use 1 multiplier, running at 2x clock rate. And other parts running at 1x clock rate.

What we expect about these versions are:

1. Multi-pumped version will save number of DSPs compared to the origin version, which will use 2 multiplier to finish 2 multiplication in one 1x clock rate cycle.
2. Multi-pumped version will be 2x faster than the resource sharing version, which will not use more multiplier, just finish the 2 multiplication one by one.

And the result looks the same:

1. **Speed**: UnrollLoop version has the fastest frequency(178MHz), and multi-pumping has similiar result(171MHz). ResourceShare version could just achieve have of the frequency(92.1MHz).
2. **Resource Usage**: For the DSPs, it's obvious that UnrollLoop will use 2 and the others two version will use one. And it should be true that Multi-pumping will have more logics usage(14479) that ResourceShare(13923) as it will have more control logic.

3.3 MaxJ SpMV Implementation(Week 5, 6)

Compressed Sparse Row(CSR) is a famous sparse matrix storage method, it pull all the non zeros to the left of their own row, and store their original column index before they have been aligned. I'm building my SpMV algorithm based on this format.

3.3.1 CSRP format introduction

In order to reduce complex control logic which could slow down the processing and introduce bugs, there'll be **padding** zeros at the end of each row, in order to make those rows, which will be processed together, have the same number of elements to calculate(although there'll be some useless zeros). That's the idea about CSRP format.

3.3.2 Redundancy formalization

If we align all our rows to the row with maximal number of nonzeros, then we could imagine there'll be lots of redundancy. As we will process only a small number of rows at the same time(number of pipes), we could just align rows to the longest row in each **processing group**.

Let's formalize this description: If we have N rows for one sparse matrix, and for row i it has L_i number of non zeros. And if we group all these rows by R, which is the number of pipes for our design, then all these R rows in one group will be calculated together. So we could devise this formula:

The lower bound of our **redundant** calculation cycles will be:

$$Redundant\ cycles = \sum_{i}^{N/R} R \times \max_{j}^{R} L_{i\times R+j} - \sum_{i}^{N} L_i \tag{1}$$

3.3.3 Redundancy Optimization

If we transform formula (1) to this form:

$$Redundant\ cycles = \sum_{i}^{N/R} (R \times \max_{j}^{R} L_{i\times R+j} - \sum_{j}^{R} L_{i\times R+j}) \tag{2}$$

So here's a heuristic rule: we need to make sure there's no such case, that a very long row and several short rows coexist in one group. According to this idea, we could simply do a **sorting** on the rows by their length(which is a greedy algorithm). Is this really the best solution? We could easily prove it.

Imagine that we have N/R "buckets", each bucket could contain R rows. And now we need to put rows inside those buckets. We suppose that the best solution is put rows one by one in the sorted sequence, which means that the rows in the first group will be the rows with top R number of non zeros.

If there's a better solution than this one, then it should have swapped some of its elements. If we swap 2 rows between 2 groups, then it will increase redundancy in both groups.

3.3.4 Performance

I've tested this format with different R value, and 2 extreme sparse matrix format: dense and triagle. The following features have been discovered:

1. The maximal performance I could get is by using design with R equals to 32, and for dense matrix, GFlops is 0.28, for triangle matrix, it's 0.22.
2. With larger R value, the performance increase at first, and the fall down.
3. If R is larger, then redundancy will be larger.

More detailed test and report will be attached once I've finished the final version.

3.4 Ruby Multi-Pumping Implementation(Week 5, 6)

In this section I'm going to make a summary about current progress on Ruby multi-pumping.

3.4.1 Ruby serialization primitives

If we want to implement multi-pumping in Ruby, the first thing we must specify is: how could we have different clock rate in Ruby? The basic primitive about clock rate changing is `pdsr` and `sdpr`. The first one will change a n length list to n cycles of one element output, which will slow down the clock rate to 1/n. The `sdpr` will perform reversely.

3.4.2 Perspectives

Just take one simple example. If we want to multi pump +1 operation, and as inc(which is +1) has only 1 input and 1 output, then we could simply decide that the input for multi pumped inc is <x, y>, a 2-list.

Here're 2 different approach:

1. **Single kernel**: we just have one kernel which runs at 2x clock rate, and 2 input streams has interleaved, 1x clock rate input.
2. **Multi kernel**: we run the kernel at 2x clock rate, but the input stream will run at 1x clock rate.

Here's an example, for the single kernel approach, the input would be like: $< x_0, \_ >, < \_, y_0 >, < x_1, \_ >, < \_, y_1 >, ...$ Please notice that this stream runs at 2x clock rate. For the second approach, we will have

$< x_0, y_0 >, < x_1, y_1 >, ...$ as input stream, and this stream will run at 1x clock rate.

The first one is simple but not so general. The second one is more accurate and harder to implement, especially for Ruby to MaxJ translation, as we need to generate manager and kernel at the same time.

3.4.3 Implementation

Here I'll introduce the solution for the second approach, which will be the test case of our Ruby to MaxJ translation function.

First we need to change the clock rate of input, which could be implemented by using `pdsr 2`. And then, do inc for each element. At last, use `sdpr 2` to bundle 2 output as 1.

Here's a single line of code: `current = pdsr 2; inc; sdpr 2.`

3.5 Ruby cyclic loop(Week 5, 6)

This part is about fixing the bug of cyclic loop in Ruby.

3.5.1 Problem: MaxCompiler auto-pipelining

For those DSP components in MaxJ, they will be automaticlly pipelined by MaxCompiler. And if we do some cyclic loop with just one offset, then this circuit is faulty as we try to get a result which is currently in the pipeline.

And if we generate the MaxJ code for `current = loop (add; DI 0; fork).`, then we will have a cyclic loop with 1 offset.

We could solve cyclic loop bug by using many solutions, but our approach is by setting the pipelining factor of this DSP to 0. We'll not have pipeline on this DSP then.

3.5.2 Solution

How could we know where exists the cyclic loop, and where to put our `push` and `pop` instructions?

For the first problem, we could detect cyclic loop by using the input and output node number of each ruby gate, generated by ruby compiler. If the output node number is less than one of the input nodes number, then there definitely exists a cyclic loop. This feature is geranteed by Ruby compiler.

For the next problem, just insert push and pop before and after the calculation part.