IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

SUMMER RESEARCH PROJECT

Multi-Pumping on Maxeler Platform

A General Study on Multi-Pumping
Implementation, Application and Compiler

Author:
Vincent ZHAO
Peking University

Supervisor:
Dr. Timothy TODMAN
Imperial College London

Prof. Wayne LUK
Imperial College London

September 8, 2015

Abstract

This research report contains both general and detailed explanation about the research project for this summer. It will start from multi-pumping, a technique which could save design resource usage without affecting the speed. And illustrate several designs which are based on, or be optimized by multi-pumping. SpMV is the most important one, it has much more aspects than those pure multi-pumping work, and it will be described in detail. At last, there's an approach to do multi-pumping by using Ruby, an HDL language which is quite simple and precise. I have finished several revision on Ruby compiler, and the work which makes the compiler to have the ability to translate Ruby into multiple kernel MaxJ code is interesting both in software algorithm and hardware design.

Contents

1 Multi-Pumping **3**
1.1 Background 3
1.2 Brief introduction to MaxJ 4
1.3 Multi-pumping MaxJ methodology 5
1.3.1 Split kernel by clock rate 5
1.3.2 Steer inputs and outputs 5
1.3.3 Configure manager 6
1.4 Experimental Study 6
1.5 Conclusion 7

2 SpMV **8**
2.1 BCSR Format 8
2.2 BCSR on FPGA 9
2.2.1 3 steps calculation 9
2.2.2 Problems 10
2.2.3 Multi-Pumping optimization 11
2.3 Experimental Study 11
2.3.1 Basic performance experiments 11
2.3.2 Multi-Pumped BCSR SpMV 12
2.3.3 Others 12
2.4 Conclusion 12

3 Ruby **14**
3.1 Cyclic loop problem 14
3.1.1 Background 14
3.1.2 Solution 15
3.1.3 Restriction 15
3.2 Serialization primitives translation 15
3.2.1 Background 16
3.2.2 Graph labeling algorithm 16
3.2.3 Kernel partitioning algorithm 16
3.2.4 Result: Multi-Pumping in Ruby 17

Appendices **19**

A Weekly Report Archive **20**
A.1 Report of Week 3 (Aug 3rd, 2015) 20
A.1.1 SpMV Implementation 20
A.1.2 Multi-Pumping 21
A.1.3 Ruby Compiler 21
A.2 Report of Week 4 (Aug 4th, 2015) 22
A.2.1 SpMV in MaxJ 22
A.3 Report of Week 5 and 6 24
A.3.1 MaxJ SpMV Implementation 24
A.3.2 Ruby Multi-Pumping Implementation 25
A.3.3 Ruby cyclic loop 26
A.4 Report of Week 7 27
A.4.1 What is BCSR? 27
A.4.2 BCSR Performance 27
A.4.3 Ruby serial primitives 28

Chapter 1

Multi-Pumping

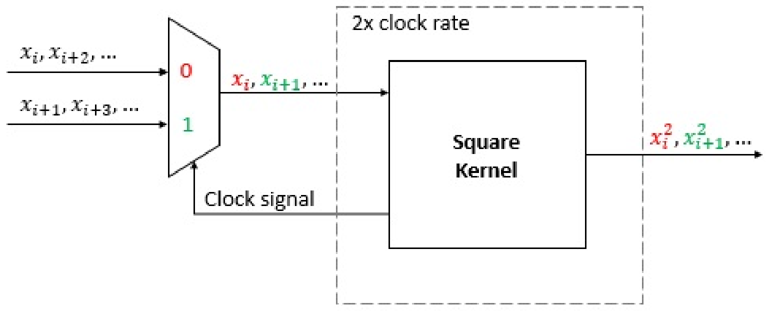
About multi-pumping, the details have been described in [1]. My work is mainly focusing on how to utilize multi-pumping technique on Maxeler platform, with MaxJ HLS language.

1.1 Background

Multi-pumping is a technique which could reduce the resource usage of a design, and in most cases, with nearly no speed affected. Multi-pumping is a problem originated from resouce sharing problem, which is mainly focusing on how to use only part of the previous resource usage without affecting the performance. And the way to solve this problem is highly dependent on scheduling, we need to schedule which operations should cooperate on one single resource. But it should be quite hard to get good resource sharing result from pure scheduling method as it's an NP-hard problem, and there're many heuristic method to optimize the algorithm's time complexity.

The basic idea of multi-pumping is to run 2 or more operations on one circuit component, with faster clock rate. For example, if we want to do 2 multiplications in one clock cycle, without multi-pumping, we need to use 2 multipliers for one cycle. But now we could run one multiplier at 2x clock rate, and obviously, this multiplier could handle 2 multiplications. Despite of this, we need one more component: multiplexer, to steer different inputs to the multiplier in different clock cycle. In the following figure, the 2x rate clock signal will be transferred to the multiplexer, and in different clock cycle, the input will be steered differenly.

Figure 1.1: A multi-pumped square circuit.



Listing 1.1: Non multi-pumped kernel example

```
1  public NoMPKernel extends Kernel {
2    NoMPKernel(KernelParameters params) {
3      super(params);
4
5      DFEVar x0 = io.input("x0", dfeUInt(32));
6      DFEVar y0 = io.input("y0", dfeUInt(32));
7      DFEVar x1 = io.input("x1", dfeUInt(32));
8      DFEVar y1 = io.input("y1", dfeUInt(32));
9
10     io.output("o1", x0 * y0, dfeUInt(32));
11     io.output("o1", x0 * y0, dfeUInt(32));
12   }
13 }
```

So multi-pumping could be regarded as an optimization to scheduling problem, from this point of view, we could implicitly schedule 2 or more operations to one resource unit, with double, triple or more clock cycles. And this is the approach from [1], they revised their own LegUp, an open source HLS system, by changing its scheduling algorithm. Their scheduling algorithm will take care of MPM(multi-pumped multiplier)'s usage.

But what if there's no way to change the underneath multi-pumping algorithm? And what if we want to do multi-pumping on much complexer components rather than simple multiplier? Like a read port of RAM, or a group of DSPs. These are my focusing points of my multi-pumping project, with Maxeler platform, I've devised a template to do multi-pumping *explicitly* on code-level, without changing the HLS algorithm. Several experiments could support the correctness of this approach.

1.2 Brief introduction to MaxJ

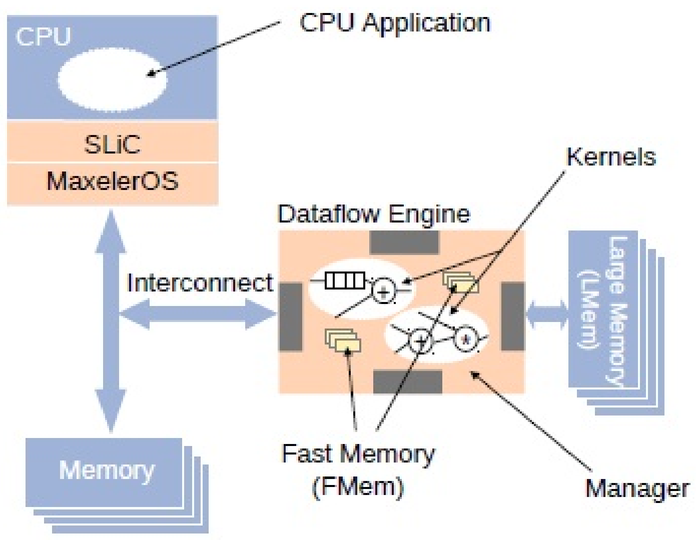
MaxJ is a programming language which supports Maxeler HLS system. It's based on Java. It's initial idea is to build hardware by running these MaxJ java code and generate a hardware configuration file. Each MaxJ project could be devided into 2 parts, MaxJ's **Kernel** will generate the hardware, and the **Manager** will handle the connections and configurations among kernels and host CPUs. Here's a simple MaxJ code snippet:

This kernel will simply multiply 4 inputs: x_0 and y_0, x_1 and y_1. And outputs them to different output streams. Each tick this kernel will take 1 element from each stream. And the manager will handle input and output connected to this kernel. It's obvious that this kernel needs 2 multipliers, as there're 2 *asterisks* on line 10 and line 11.

More precisely, the generated output of MaxJ is called DataFlow Engine(DFE, see Figure 1.2), which take inputs and outputs as streams, and process them in a dataflow pattern. Kernels that generated could be regared as processing units in the dataflow, and manager handles how the dataflow connected to each other. MaxJ generate a configuration file of Maxeler DFE which is called Maxfile, which could be used to generate interface between host CPU and DFE, and set up bitstreams on FPGA hardware.

In the next sections, we'll focus on 1 problem: how could we transform a non multi-pumped kernel code in MaxJ, to a multi-pumped one? It seems not so hard at first glance, but there do have some tricky details we need to take care of. And by using the methodology I've devised and illustrated later, we could build multi-pumping project in the code level systematically

Figure 1.2: DFE architecture



1.3 Multi-pumping MaxJ methodology

There're 3 steps to transforma non multi-pumped kernel to a multi-pumped one:

1. **Split kernel by clock rate**: we put components with different clock rate into different kernels.
2. **Steer inputs and outputs**: In kernels with faster clock rate, we need to steer input data in different clock cycles.
3. **Configure manager**: Including clock frequency configuration and kernel connection details.

1.3.1 Split kernel by clock rate

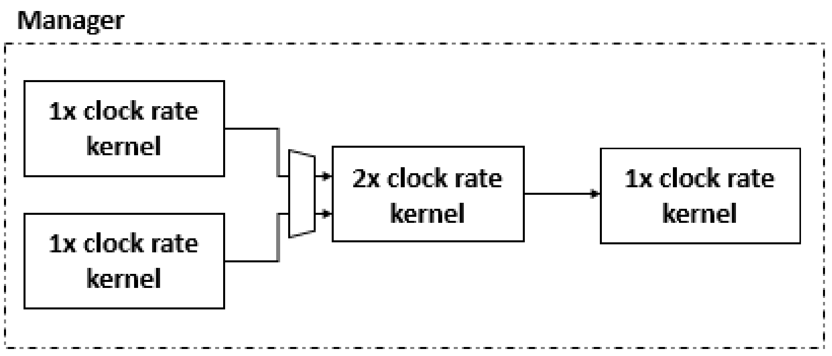
In order to write Multi-pumping code explicitly, you need to assign different clock rate for different components. And according to MaxJ's convention and restriction, one kernel could only have one clock rate, which could be set in manager. If you have an original MaxJ design which only contains one kernel, the first thing you need to do to multi-pump it is splitting it into multiple kernels, and different kernel has different clock rate.

We could describe our multi-pumping design in a DFE style(Figure 1.3). There're several 1x clock rate kernel handles slow operations, and take or push input to 2x clock rate kernel. This 2x clock rate kernel could have contained many DSPs, or some BRAMs with read or write ports, or some other fast components.

1.3.2 Steer inputs and outputs

We could do this in the manager level, by using `mux` method of manager class. However, it's more intuitive to do it in kernel, by using simple counter. We could revise our original Non multi-pumped code like this:

Figure 1.3: DFE Multi-Pumping



Listing 1.2: Non multi-pumped kernel example

```
DFEVar counter = control.count.simpleCounter(2);

DFEVar x0 = io.input("x0", dfeUInt(32), counter === 0);
DFEVar y0 = io.input("y0", dfeUInt(32), counter === 0);
DFEVar x1 = io.input("x1", dfeUInt(32), counter === 1);
DFEVar y1 = io.input("y1", dfeUInt(32), counter === 1);

io.output("o1", x0 * y0, dfeUInt(32), counter === 0);
io.output("o1", x0 * y0, dfeUInt(32), counter === 1);
....
```

Here === could be regarded as "take value and compare with". For the counter, it will count from 0 to 1 and loop back to 0. So it's equivilant to use a multiplexer which take clock signal as input. And the output will also be controlled by this counter value.

1.3.3 Configure manager

The most important thing to configure in manager is the clock rate. Under MaxJ's context, all the clock rate will be set only for stream, and the kernel clock rate could be derived from its in/out stream clock setting. For example, if the input stream of one kernel is 100MHz, then this kernel will run as 100MHz. It follows the dataflow nature of Maxeler DFE platform.

And also, currently there's no way to specify ratio timing constraints, such as one stream is 2 times faster than the other stream, so we need to set clock value explicitly: global stream is 100MHz and the multi-pumped kernels' stream will be 100n MHz.

1.4 Experimental Study

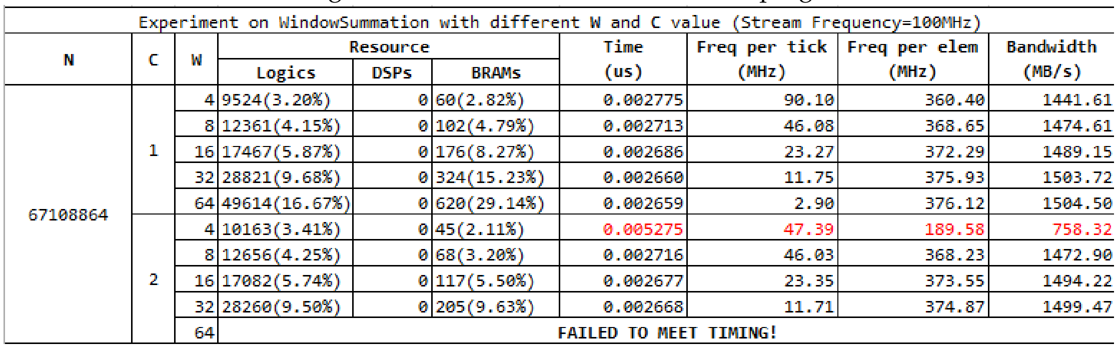
Their are 2 main studies of multi-pumped MaxJ code: One is called *inner square product*, which will do $x_i^2 \times y_i^2$ for 2 input streams, the other one is called *window summation*, which calculates summation of a window of array, and the array's value will be gathered from ROM. The latter one could be written in:

$$Sum = \sum_{i=0}^{W} rom[index[i]]$$

Where W is the window width, and $index$ is the input stream which contains address of rom to read. The code of these 2 project could be referenced from this repository [1], and the experimental data has also be included there.

For the first design, it's focusing on integrating kernel with many DSPs to one multi-pumped kernel. And for the second one, it shows a noval implementation of multi-pumping on ROM read, which could save a lot of memory space on board as one copy of BRAM could only support 2 read ports at the same time. With multi-pumping, we could do 4, 8 and more(although this may not be a good idea) read ports in one tick. In this figure, the multi-pumped version(C=2) has nearly half of the BRAM resource usage than non multi-pumped one(C=1), and the frequecies are nearly the same.

Figure 1.4: ROM Read Port Multi-Pumping



Experiment on WindowSummation with different W and C value (Stream Frequency=100MHz)

N	C	W	Resource			Time	Freq per tick	Freq per elem	Bandwidth
			Logics	DSPs	BRAMs	(us)	(MHz)	(MHz)	(MB/s)
67108864	1	4	9524(3.20%)	0	60(2.82%)	0.002775	90.10	360.40	1441.61
		8	12361(4.15%)	0	102(4.79%)	0.002713	46.08	368.65	1474.61
		16	17467(5.87%)	0	176(8.27%)	0.002686	23.27	372.29	1489.15
		32	28821(9.68%)	0	324(15.23%)	0.002660	11.75	375.93	1503.72
		64	49614(16.67%)	0	620(29.14%)	0.002659	2.90	376.12	1504.50
	2	4	10163(3.41%)	0	45(2.11%)	0.005275	47.39	189.58	758.32
		8	12656(4.25%)	0	68(3.20%)	0.002716	46.03	368.23	1472.90
		16	17082(5.74%)	0	117(5.50%)	0.002677	23.35	373.55	1494.22
		32	28260(9.50%)	0	205(9.63%)	0.002668	11.71	374.87	1499.47
		64	FAILED TO MEET TIMING!						

1.5 Conclusion

Multi-pumping as a resource sharing technique, could be supported in HLS system originally, and also it has been discovered that in code-level we could also use multi-pumping to save space harmlessly. The methodology about implementing multi-pumping on MaxJ could serve as code template and it's possible to use this template to multi-pump any kinds of "faster" components. It has been proved that kernel with multiple DSPs and ROM read ports could be multi-pumped seamlessly.

In the next chapter, I will mention this multi-pumping practice on SpMV, a much complexer problem then these 2 examples, about how could we find the resource bottleneck and how to reduce them by using multi-pumping. In the last chapter, this multi-pumping technique will be described and implemented in 2 simple serialization primitives, in Ruby HDL programming language. And the most important part is we could generate practical multi-pumped MaxJ code from this *theoretical* HDL language.

[1]Github repository folder: `https://goo.gl/uxJ9x0`

Chapter 2

SpMV

The focus of this SpMV project is not to provide the fastest SpMV implementation for some specific purpose, like what most other SpMV papers do. Here this project will have a general purpose implementation, which could handle most sparse matrix cases, with pretty good performance(May not be fastest though). The format of sparse matrix that I choose to use is called Blocked Compress Sparse Row(BCSR), which generates blocks upon CSR format. This chapter I'll first describe what is BCSR format and my MaxJ implementation of this format, experimental data will also be shown here. At last, we'll integrate our multi-pumping techniques with this SpMV implementation to see whether we could get better resource usage.

2.1 BCSR Format

This format could be generalized as a blocked version of CSR format. To generate a CSR representation of a sparse matrix, we could simply align the non zero elements to the left(or the other side if you wish). Here's an example sparse matrix, where those letters are representing non zero matrix elements.

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

And if we want to build a CSR represenation of this matrix, we could just pull all the non zero elements to the left.

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

It depends on yourself about how to store the CSR format, basically, it's just a 2 dimensional array, with each element as a pair of value and original column index. For the previous CSR example, the storage format could be:

$$\begin{bmatrix} (a,0) & (b,2) & (c,3) \\ (d,0) & (e,1) & (f,2) \\ (g,3) & & \\ (h,1) & (i,2) & \end{bmatrix}$$

These format works for many different matrices, and platforms, but its main deflect is that it will not be padded, and this is obviously not good for parallel processing units, especially our DFE design. So we choose to use BCSR format, which will pad zeros to each row, to align and group non zeros and padding zeros to blocks. The BCSR format with 2X2 block of the previous matrix is:

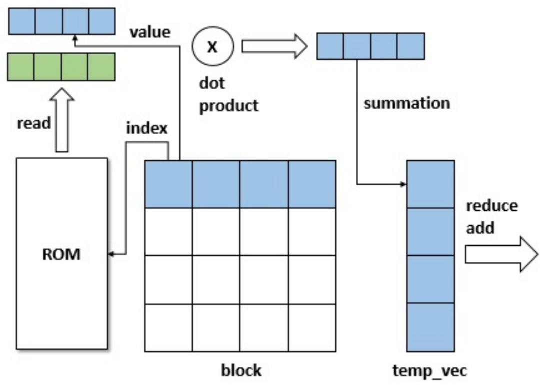
$$\begin{bmatrix} (a,0) & (b,2) & (c,3) & (0,0) \\ (d,0) & (e,1) & (f,2) & (0,0) \\ (g,3) & (0,0) & & \\ (h,1) & (i,2) & & \end{bmatrix}$$

With this format, we could simply pass one block to DFE in one tick. The further question is, how could we process $R \times C$ shape block in one tick?

2.2 BCSR on FPGA

The first thing we must specify is about where to put the data. Although Maxeler board has a lot of BRAMs to use, it will still be insufficient if we simply put all the storage on board. The simplest approach is to put vector value on BRAMs, and set input streams to take BCSR's index and value streams. So for each tick, each input stream will take $R \times C$ number of elements if the block has been configured as $R \times C$ shape.

Figure 2.1: BCSR one block computation



2.2.1 3 steps calculation

For each tick, the BCSR implementation will mainly do 3 things:

1. **Read from ROM**: For each row in the block, we use original column index to read value from vector, which has been stored on ROM.
2. **Multiply**: After read a size C vector from ROM, we do multiplication between this little vector and the value from the row we're calculating. You could view this as the first step of dot product.
3. **Summation**: Then we do the second step of dot product, sum up the result vector we get from the previous step.

If we have multiple rows to calculate in one tick, then we apply these 3 steps in parallel. And here's a wireframe of 4 X 4 block configuration.

Of course, we could describe this design in MaxJ kernel code(Listing 2.1). In this code snippet, the function `summationTree` will generate a tree-like summation circuit, which has $log_2(C) - 1$ level of adders, and in level $i = 0...log_2(C) - 1$ will contains 2^i adders.

Listing 2.1: BCSR kernel

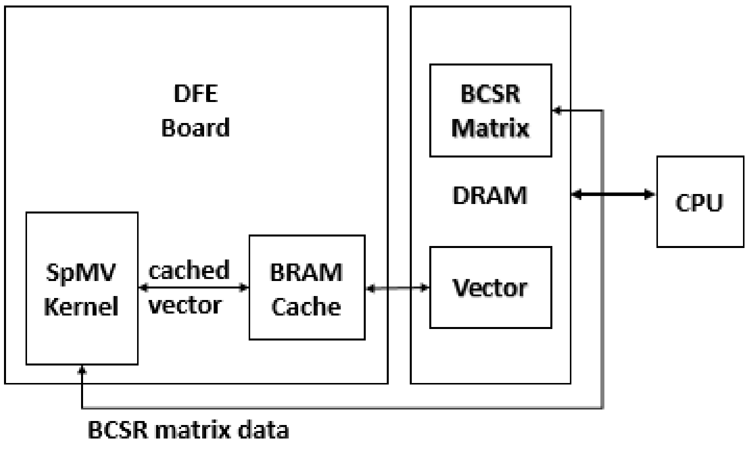
```
for (int i = 0; i < R; i ++) {
  // gather
  for (int j = 0; j < C; j++)
    vectorValue[j] <== ROM.read(index[i*R+j]);
  // multiply
  for (int j = 0; j < C; j++)
    productValue[j] <== vectorValue[i] * rowValue[j];
  // summation
  result[i] <== summationTree(productValue, 0, C-1);
}
```

And each line could have multiple blocks, we need to sum up the `temp_vec` to get the final result for each row, which is initially a reduce add operation.

2.2.2 Problems

This BCSR design looks good, but it has several deflects: First, it can't handle large sparse matrix computation. As large matrix has large column size, which means we need to store a large amount of vector data inside ROM. And also, as we need to do multiple read in one tick for each block($R \times C$ times), we need to have $\frac{R \times C}{2}$ number of ROM copies as each ROM could only support 2 read ports in one tick. That's a huge amount of data and there's no sufficient place for these BRAMs. And also, as we're streaming matrix data in, the maximal bandwidth of PCIe will restrict our performance as our design could always process data elements faster.

Figure 2.2: BCSR new architecture



So the main problem is about storage and bandwidth, and the major solution to this problem is by using LMem, a outside DRAM for DFE, which has large storage and connect with DFE with 20x more bandwidth. As LMem do not support random access, which is quite essential for

ROM read of SpMV, we use an on-board BRAM cache to handle random access(Although this need more reordering on matrix). Here's the optimized architecture of our BCSR design(Figure 2.2).

2.2.3 Multi-Pumping optimization

If you take a closer look at our 3 steps BCSR calculation, you could find out that there're 2 steps that match our multi-pumping examples: one is **gather**, where we could multi-pump the ROM read, and **multiply**, those DSPs could be multi-pumped. Our BCSR SpMV will benefits a lot from this: It uses many ROM to do multiple reading and many DSPs to do dot production, with multi-pumping we could highly reduce those resource usage. And also, the **summation** operation is quite slow if we apply block configuration with large C value, which means our multi-pumped components could run at more than 2x clock rate of that slow clock rate.

This part of work is still under evaluation, and may not be released until everything works fine.

2.3 Experimental Study

Experiments on this SpMV application are focusing on two aspects: One is about the maximal speed we could get, and some discovery about different parameters' could affect the performance. The other one is on the resource usage of our multi-pumping application, mainly on DSPs usage, we try to find out whether the resource usage will be reduced by using our multi-pumping design.

2.3.1 Basic performance experiments

According to our algorithm and design, if there's no bandwidth limitation(or comparing to our speed, its much faster), then the performance should be estimated as:

$$\text{Estimated Frequency} = \text{Frequency} \times R \times C$$

So if we use R and C with the same $R \times C$ value, then the performance shouldn't be different. We evaluate the performance with 3 different metrics, suppose the runtime for each non-zero element is T:

1. **Bandwidth**: Bandwidth is the number of bytes that has been transferred in one second. In our application, on BCSR non zero pair will take 6 bytes, as 4 bytes for float value and 2 bytes for `uint16_t` type index.

$$\text{Bandwidth} = \frac{6}{T} \times 10^{-6} (\text{MB/s})$$

2. **GFlops**: The effective floating point operation number will be two for each non zero element, one is for adder, the other is for multiplier.

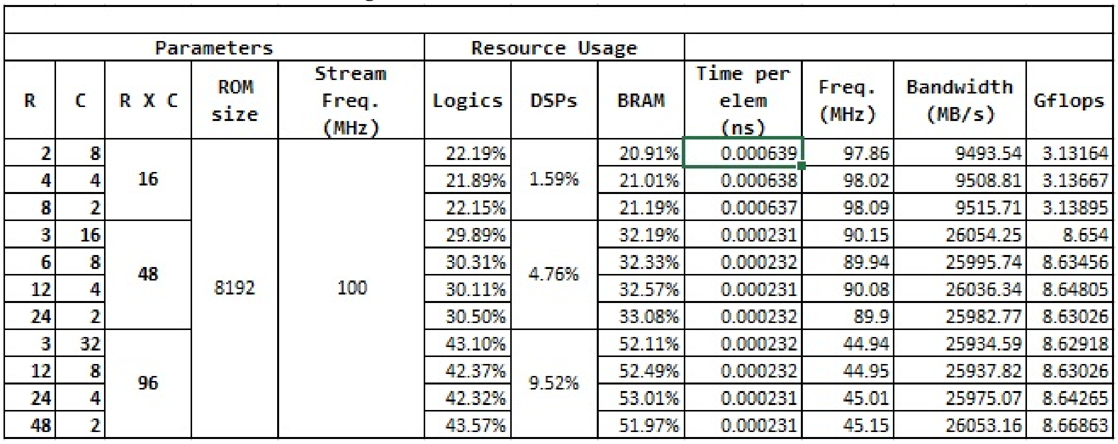
$$\text{GFlops} = \frac{2}{T} \times 10^{-9}$$

3. **Frequency**: This is the real frequency we get from runtime. It should be close to our estimated frequency, although there will be some interferer while calculating.

$$\text{Frequency} = \frac{1}{T} \times 10^{-6} (\text{MHz})$$

Here're the results we've got. We have tested 3 sets of $R \times C$ configurations: 16, 48 and 96. The reason we choose last 2 number is for the transfer between LMem and DFE(or CPU) should be divided by 384 bytes. And as C must be a number as power of 2, as the summation tree is a full binary tree, we need to fullfill every level. These data are based on several self-generated sparse matrices with number of columns as 8192 and total datasize equals to 8192^2. We have generated dense matrices, triangle matrices, band matrices and random matrices. The performance result following are average results.

Figure 2.3: BCSR LMem Performance



Parameters					Resource Usage							
R	C	R X C	ROM size	Stream Freq. (MHz)	Logics	DSPs	BRAM	Time per elem (ns)	Freq. (MHz)	Bandwidth (MB/s)	Gflops
2	8	16	8192	100	22.19%	1.59%	20.91%	0.000639	97.86	9493.54	3.13164
4	4				21.89%		21.01%	0.000638	98.02	9508.81	3.13667
8	2				22.15%		21.19%	0.000637	98.09	9515.71	3.13895
3	16	48			29.89%	4.76%	32.19%	0.000231	90.15	26054.25	8.654
6	8				30.31%		32.33%	0.000232	89.94	25995.74	8.63456
12	4				30.11%		32.57%	0.000231	90.08	26036.34	8.64805
24	2				30.50%		33.08%	0.000232	89.9	25982.77	8.63026
3	32	96			43.10%	9.52%	52.11%	0.000232	44.94	25934.59	8.62918
12	8				42.37%		52.49%	0.000232	44.95	25937.82	8.63026
24	4				42.32%		53.01%	0.000231	45.01	25975.07	8.64265
48	2				43.57%		51.97%	0.000231	45.15	26053.16	8.66863

For the performance, we could reach a maximal GFlops around 8.6, and there's a sharp frequency drop when we tried to use $R \times C = 96$, which might meet the boundary bandwidth of LMem to DFE. The resource usage looks quite reasonable: The DSP usage is highly related to the $R \times C$ value, and so does the BRAM usage.

2.3.2 Multi-Pumped BCSR SpMV

Currently we have only tested the $R = 4$ and $C = 4$ with double pumped configuration. The one without multi-pumping will use 32 multipliers for calculation while the multi-pumped will only use half of them.

2.3.3 Others

Other experimental data are stored in a spreadsheet, which contains the studies of ROM optimization, LMem optimization and bandwidth optimization of our SpMV BCSR algorithm. And also it includes those data which related to the previous CSRP format, the one that has been abandoned.

2.4 Conclusion

Our BCSR SpMV implementation could handle most of the sparse matrix cases, it do not depends on the distribution of number of non zeros in each row, and with the help of LMem and multi-pumping, we could store very large vector for calculation. Multi-pumping here not only reduces the DSP usage, but also reduce a large amount of BRAM usage, which is the resource bottleneck of our SpMV design.

The way we apply multi-pumping on a non multi-pumped SpMV design suggests that our multi-pumping methodology will work perfectly, and could adapt to many different circumstances.

Chapter 3

Ruby

Ruby is a highly abstract HDL language, which could write a very complex design in quite a few lines. Comparing to MaxJ, ruby code is much simpler and could help programmer concentrate on the design itself, without distracting on allocating memory, configuring and connecting input and output streams. But ruby don't have a powerful toolchain like MaxJ, which could use single command to deploy your hardware design on Maxeler FPGA board. So it's a good idea to combine these 2 languages together, with Ruby's simple syntax and MaxJ's toolchain. We could achieve this simply by translating Ruby code into MaxJ code, and do the compilation.

But as Ruby is highly abstract, there's no explicit configurations in the source code file, and Ruby itself doesn't contain optimization of the design, like what MaxJ do underneath. There'll be a conflict when MaxJ do pipelining implicitly, and Ruby wants to do cyclic loop on the pipelined operator. This problem will be covered later, and the solution will also be listed.

Ruby provides serialization methods, which could convert clock rate between the input and output ports of wires. If we simply translate one Ruby file into one MaxJ kernel file, then there'll be problem, as we have mentioned: MaxJ do not support multiple clock configuration on one kernel. And Ruby doesn't have kernel splitting primitives, we need to do this automatically. There's an algorithm that has been implemented to do this when translating Ruby code to MaxJ, the main idea is to split by clock rate changing, especially by `pdsr` and `sdpr` primitives.

We could also use these 2 primitives to implement multi-pumping in Ruby. There'll be an example about how to do this in Ruby, and an example of translated MaxJ kernels and manager files could prove that our kernel splitting algorithm works.

3.1 Cyclic loop problem

3.1.1 Background

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. There'll be problem if this `add` has been pipelined. The 1 offset before result we get are still in the pipeline when we get it.

Could we solve this problem without affecting the pipelining optimization? The answer is no. It's quite hard to generate a C-Slowed MaxJ code directly from Ruby, and without C-Slowing, the generated code will be really slow, as we are running the code with $\frac{1}{pipeline\ depth}$.

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.

3.1.2 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.

Here's the code snippet related to this solution:

Listing 3.1: Revision to fix cyclic loop

```
(pushPipelineIfCyclic (inNodes, outp)) ^
(showGate (maxBackPatchForwardDeclaration) (device, input, output)) ^
(popPipelineIfCyclic (inNodes, outp))
```

The `showGate` function is the one output the string of MaxJ's assignment instructions, and we wrap this up if there's a cyclic loop within. Those `pushPipelineIfCyclic` or pop functions first detect the input nodes of this current gate, if this gate contains input nodes `inNodes` that is larger than the output nodes `outp`, we will output a string that push/pop pipelining factor 0.

Listing 3.2: Implementation of cyclic loop detector

```
fun isCyclicLoop(inNodes, outp) =
    List.foldl (fn (x, b) => ((x > outp) orelse b)) true inNodes

fun pushPipelineIfCyclic (inNodes, outp) =
    if isCyclicLoop(inNodes, outp)
    then "␣␣␣optimization.pushPipeliningFactor(0);\n"
    else ""

fun popPipelineIfCyclic (inNodes, outp) =
    if isCyclicLoop(inNodes, outp)
    then "␣␣␣optimization.popPipeliningFactor();\n"
    else ""
```

3.1.3 Restriction

As I've mentioned before, this solution is far from perfect. We simply banned pipelining, and the direct result would be a tremendously low maximal frequency we could achieve, especially we want to do complex design.

3.2 Serialization primitives translation

Most of the ruby's serialization primitives work fine while we're translating them from ruby to MaxJ, for example, `DI` the delay unit will be translated to `stream.offset` as what we expect. However, there're 2 more primitives that can't be correctly translated: `pdsr` and `sdpr`.

3.2.1 Background

`pdsr n` is a primitive which takes a n list of elements, and output 1 by 1 in the following n cycles. And `sdpr n` do things in reverse, they take n elements and output a n list in one cycle. The reason why this is not supported is that, the current ruby to MaxJ translation code could only support one kernel generation, `pdsr` and `sdpr` need to change the clock rate between those input and output ports. As we know that we could only assign one clock rate for one kernel in MaxJ, so it's impossible to do this if there's no multiple kernels generation mechanism supported.

Things are more difficult as there's no primitives that support kernel partition functionality. Programmers couldn't assign which part of the ruby circuit should be in one kernel. So we should choose whether to revise the ruby syntax and compiler itself, or detect kernels by some heuristic methods. We choose the second one.

3.2.2 Graph labeling algorithm

Currently, the main purpose of our kernel partitioning is due to clock rate changing while we're translating serialization ruby primitives to MaxJ. It's obvious that we should first(and maybe the last) to detect kernel border by using clock rate. So first of all, we need to generate a graph, or more precisely, a DAG that contains clock rate information.

We build a graph from the raw output from ruby compiler first, which is called `pcircuit`. It contains the basic information of one ruby file. This data structure contains 3 components: 2 expressions of domains and ranges, and the relations in ruby code. We should know that there're 2 levels of circuit organization in ruby: The first level is called gate, which contains input and output ports, and one single processing unit(sort of). And the second level is called relation, which contains many gates, and all these gates are connected in parallel, which means they will not have dependencies to each other. The expression is another special type `expr`, it contains many constructors, but we mainly use the `WIRE` to get the wire id.

Start from input nodes, we could "flow" from input to output of the ruby file, and during this flowing procedure, we could met `pdsr` or `sdpr`, and they will change clock rate from input to output. So if we build have a initialized clock for input wires, and change or remain the same clock rate as the previous wire, then we could build a global graph labeled with clock rate.

Listing 3.3: Build labeled graph

```
val (eL1,iL1) = appendEdgesToEdgeList (exprToEdges dom) []
val (eL2,iL2) = appendEdgesToEdgeList (exprToEdges ran) eL1
val g         = appendRelsToGraph rels ([], eL2)
val setClk    = initGraphClock 1.0 g
```

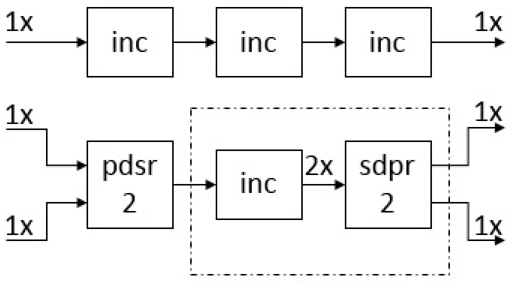
This code snippet lies in the main procedure of our kernel partitioning algorithm, as the initilization steps. It first append input and output nodes to the graph, and then append relation to build the edges between nodes. At last it will initialize the clock rate for each edge.

3.2.3 Kernel partitioning algorithm

Then we could partition our built graph. As we intends to split the graph by using different clock rate as boundary, the we could start from the input nodes, and mark all those edges which contains different clock rate in input edges and output edges as **boundary**. The tricky point of this algorithm is, it will stop immediately when we met one different clock rate node, just like a wavefront.

Here's a diagram to show this(Figure 3.1). The components in the dashed rectangle should be partitioned as a new kernel. And all the components outside the 2x clock rate box should be integrated in one.

Figure 3.1: Diagram of kernel partitioning



This is the code snippet to do this. It's prefixed with iter as this function will iteratively build the graph. It takes a list of input nodes, and start with input nodes to take all those edges and nodes it met as a graph. It will stop in 2 circumstances: output or met clock rate border. The history list will store those input nodes it has met to avoid infinite loop.

Listing 3.4: Iter partitioning algorithm

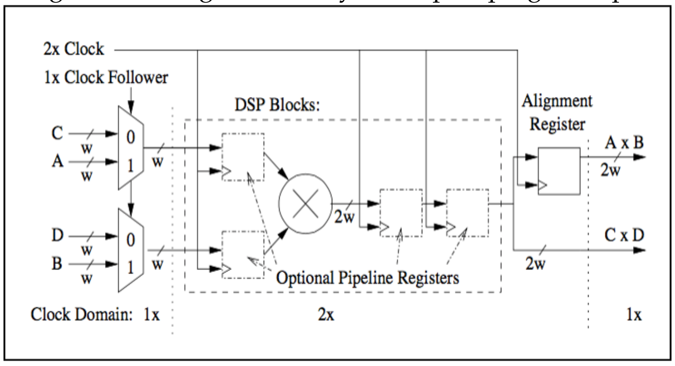
```
fun iterSplitGraph [] hs g = []
    | iterSplitGraph es hs g =
    ( let
        val splitEdgeIds = iterFindSplitEdges es g
        val splitNewGraph= splitAndFilterEdgesInGraph splitEdgeIds g
        val newInEdgeIds = getGraphInputEdges splitNewGraph
        val nextHistory  = es @ hs
        val nextInEdgeIds=
      filterExistEdgeList
        (getListCompliment nextHistory newInEdgeIds)
        splitNewGraph
        val build        = iterBuildGraph es splitNewGraph ([],[])
        val retGraphList = iterSplitGraph nextInEdgeIds nextHistory splitNewGraph
      in
        build::retGraphList
      end )
```

3.2.4 Result: Multi-Pumping in Ruby

Besides this, we also need some functions to support manager and cpu code printing. I'll not mention them here, but it should be mentioned that the printer function works fine and we could run MaxJ simulation directly from ruby file(by using some Makefile configuration of course).

Here's an generated example of ruby multi-pumping and the generated MaxJ code. Here we will multi-pump one multiplier, make it serve 4 input streams, just like this example(Figure 3.2). If we ignore those pipeline registers and multiplexers, we could build this circuit by using this ruby code:

Figure 3.2: Diagram of ruby multi-pumping example



```
current = [pdsr 2, pdsr 2]; mult; sdpr 2.
```

Here're the generated MaxJ code. There'll be 2 kernel files generated, the first one only handles the input stream and by using pdsr, we could merge these 4 streams in 2 streams. Then the second kernel will run at 2x clock rate to take the input streams, and do multiplication.

Listing 3.5: Generated MaxJ code

```
public class CurrentManager extends CustomManager {

public CurrentManager (EngineParameters params) {
   super(params);

   // kernel block declarations:
   KernelBlock k0 = addKernel(new Kernel1(makeKernelParameters("Kernel1")));
   KernelBlock k1 = addKernel(new Kernel2(makeKernelParameters("Kernel2")));

   // kernel block clocks declarations:
   float clk = 100;
   config.setDefaultStreamClockFrequency((int)clk);
   ManagerClock clk0 = generateStreamClock("clk0", (int)(clk * 1.0));
   k0.setClock(clk0);
   ManagerClock clk1 = generateStreamClock("clk1", (int)(clk * 2.0));
   k1.setClock(clk1);

   // DFELinks declarations:
   addStreamToCPU("board2cpu_0") <== k1.getOutput("out4");
   addStreamToCPU("board2cpu_1") <== k1.getOutput("out3");
   k1.getInput("inp6") <== k0.getOutput("out2");
   k1.getInput("inp5") <== k0.getOutput("out1");
   k0.getInput("inp4") <== addStreamFromCPU("cpu2board_0");
   k0.getInput("inp3") <== addStreamFromCPU("cpu2board_1");
   k0.getInput("inp2") <== addStreamFromCPU("cpu2board_2");
   k0.getInput("inp1") <== addStreamFromCPU("cpu2board_3");

   createSLiCinterface(interfaceDefault());
} // end constructor
```

Appendices

Appendix A

Weekly Report Archive

A.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.

A.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)

A.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.

A.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.

A.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.

A.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.

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.

A.3 Report of Week 5 and 6

A.3.1 MaxJ SpMV Implementation

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.

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.

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:

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

Redundancy Optimization

If we transform formula (1) to this form:

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

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.

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.

A.3.2 Ruby Multi-Pumping Implementation

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

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.

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: $< \mathtt{x}_0, \_ >, < \_, \mathtt{y}_0 >, < \mathtt{x}_1, \_ >, < \_, \mathtt{y}_1$ Please notice that this stream runs at 2x clock rate. For the second approach, we will have $< \mathtt{x}_0, \mathtt{y}_0 >, < \mathtt{x}_1, \mathtt{y}_1 >, ...$ 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.

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.`

A.3.3 Ruby cyclic loop

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

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.

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.

A.4 Report of Week 7

A.4.1 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.

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:

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

A.4.2 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:

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

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.

A.4.3 Ruby serial primitives

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

`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.

`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`.

Bibliography

[1] Canis, A., Anderson, J. H., and Brown, S. D. Multi-pumping for resource reduction in fpga high-level synthesis. In *Proceedings of the Conference on Design, Automation and Test in Europe* (San Jose, CA, USA, 2013), DATE '13, EDA Consortium, pp. 194–197.