This is the Author-Submitted Version of the paper:

J. Vreča and A. Biasizzo, "A Configurable Mixed-Precision Convolution Processing Unit Generator in Chisel," *2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, Tallinn, Estonia, 2023, pp. 128-131, https://doi.org/10.1109/DDECS57882.2023.10139758.

A Configurable Mixed-Precision Convolution Processing Unit Generator in Chisel

1st Jure Vreča
Computer Systems Department (E7)
Jožef Stefan Institute
Ljubljana, Slovenia
jure.vreca@ijs.si

2nd Anton Biasizzo
Computer Systems Department (E7)
Jožef Stefan Institute
Ljubljana, Slovenia
anton.biasizzo@ijs.si

Abstract

We present a configurable implementation of a convolution processing unit suitable for computing mixed-precision quantized neural networks. We used Chisel to write the hardware generator, it is a framework for writing hardware circuit generators. Our generator is designed to use minimal hardware resources and is flexible in various aspects of the convolution operation, including: image size, kernel size, image bitwidth, kernel bitwidth, activation function, and more. The processing unit is static after generation, thus we don't pay the price of using more general hardware, instead we can tailor it to the problem at hand.

Index Terms

neural networks, quantization, Chisel, FPGA

I. Introduction

Neural networks have proven useful in many areas, but they are very computationally intensive, and are difficult to implement on low-power devices. The root cause of their processing complexity is the vast number of parameters, and consequently their memory footprint. For example AlexNet uses 60 million floating-point parameters, which use around 240 MB of memory [1]. However, research has shown that using floating-point parameters is not needed to achieve good accuracy of a neural network. In fact, satisfactory results cay be achieved with using fixed-point arithmetic with bitwidths as low as only a single bit [2, 3]. Such networks are referred to as quantized neural networks.

While quantization decreases accuracy of networks, not all layers of the neural network are equally sensitive to quantization. Therefore, some layers may be quantized to lower bitwidths than others, while still preserving accuracy of the overall neural network. Such an approach is called mixed-precision quantization, and it can achieve a more optimal memory footprint to neural network accuracy ratio [4, 5].

Unfortunately, most modern hardware is unable to properly take advantage of lower bitwidth formats, as it is build for computing with larger words, like 64- or 32-bit. Instead of designing hardware to handle arbitrary bitwidths dynamically, we opted to design a generator that can generate the required static processing unit, that is tailored to a specific convolution layer. A set of such processing units are connected together in series to compute the entire convolutional neural network. Such an approach is not completely novel [6, 7]. However, previous work mostly focused on using high-level synthesis (HLS) tools, which can produce less optimal results [8]. We present an alternative approach of developing such hardware in the Chisel Hardware Construction Language (HCL) [9].

II. The Chisel HCL

Chisel is not a Hardware Description Language such as Verilog or VHDL, instead a software program is developed, that generates digital logic. Internally Chisel first elaborates the design to an intermediate representation, that can then be exported as synthesizable Verilog. Chisel is implemented as a framework in the Scala programming language. This means that you can use many advanced features of the Scala programming language, like object oriented and functional programming, to build your hardware. Chisel also has a testing facility called ChiselTest, which enable the verification of the circuit.

All hardware characteristic in Chisel are represented as Scala classes. For example, a module is defined as a child of an abstract class $Module$, which has an IO field (an object), that defines the input/output interface of the module. Chisel is a typed language, so we must provide each signal with a Chisel type. This can be $Bool$ for boolean values, $UInt$ for unsigned integers, $SInt$ for signed integers, and there also exist some other less often used types. When defining some types, like $UInt$, we can also add a width parameter, that is denoted with the $.W$ postfix. However, not all widths need to be specified, as some can be inferred from the code. To illustrate the Chisel framework a simple example of a Chisel module, that generates an incrementer is given in Listing 1.

The authors acknowledge the financial support from the Slovenian Research Agency (research core funding No. P2-0098). This work is also part of a project that has received funding from the ECSEL Joint Undertaking under grant agreement No 101007273 (DAIS).

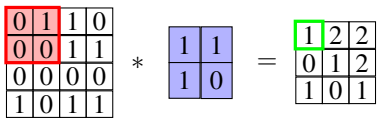

Fig. 1. The convolution operation.

One of Chisel most useful features is its ability to parameterize a design. Listing 2 shows a parameterized version of an incrementer that has one input and one output of type T. The expression $T <: Bits\ with\ Num[T]$ is a type bound expression which forces type T to be a sub-type of the Chisel base type $Bits$, and this type must implement the Num trait, which limits the set of types to the types which have defined the addition operator. Such a parameterization allows us to generate modules that add the constant n to the inputs of various types. Chisel also accepts functions as parameters, and even entire modules.

```
class Inc1 extends Module {


 val io = IO(new Bundle{
   val in = Input(UInt(3.W))
   val out = Output(UInt())
 })
 io.out:=io.in + 1.U
}
```

Listing 1. Simple design.

```
class IncN[T<:Bits with Num[T]](
    t:T,
    n:Int) extends Module {
  val io = IO(new Bundle{
    val in = Input(t)
    val out = Output(t)
  })
  io.out:=io.in+n.S.asTypeOf(t)
}
```

Listing 2. Parameterized design.

III. Processing Unit

A. The Convolution Operation

Our generator generates a circuit tailored to a particular quantized convolution layer. The mathematical operation that our processing unit performs can be expressed as an operation on a four-dimensional weight tensor w (a set of kernels), and a three-dimensional input tensor x. The result is another three-dimensional tensor y. The dimensions of w are K, C, F, F; where K is the number of kernels, C is the number of channels in each kernel, and F is the width and height of the kernel, which we constrain to be equal. The dimensions of tensor x on the other hand are C, W, H. The channel dimension C of tensor x must be equal to the channel dimension of tensor w, and both W and H must be greater than F. A computation of a single element of the output y is shown in Eq. (1),

$$y[k,i,j] = f(\sum_{c=0}^{C-1}\sum_{x=0}^{F-1}\sum_{y=0}^{F-1} x[c, i+x, j+y] \cdot w[k,c,x,y] + b) \tag{1}$$

where f is an activation function, b is the bias value, and k identifies the used (applied) kernel as well as the output channel. Fig. 1 shows a small example of a convolution operation, where $C = 1$, $K = 1$, $F = 2$, $W = H = 4$.

The accuracy of quantized neural networks can be increased by scaling the output. In efficient implementations the scaling factors are limited to powers-of-two. This simplifies the scaling operation to a shift operation, as described in [10].

B. Hardware Architecture

Fig. 2 shows the block diagram of our processing unit. It contains the following units:

- three memory units (local static RAM):
  - FEATMEM, a memory unit to store the input feature map,
  - KERNELMEM, a read-only memory initialized with the kernel weights,
  - RESMEM, the result memory unit.
- Two register files: the first for kernel parameters, and the second holds a subset of the input feature map.
- A Neuron Compute Unit.
- A Threshold/Shift Unit.
- Kernel RF Loader Unit, Sliding Window Unit, and other control logic.

In the next subsections, we will discuss some of these hardware units in more detail.

C. The Neuron Compute Unit

An abridged version of the Neuron Compute Unit is shown in Listing 3. It is written in a generic form in Chisel and can be used to generate hardware for various different quantization parameters. The module is type parameterized with five type parameters, namely: the input type I, the weight type W, the multiplicand type M, the threshold type A, and the output type O. These type parameters can be either $UInt$ or $SInt$ of a specified bitwidth, and thus we can set the bitwidth of the inputs, the weights, the multiplicands, the accumulator and the output. It also accepts the activation function, multiplication function

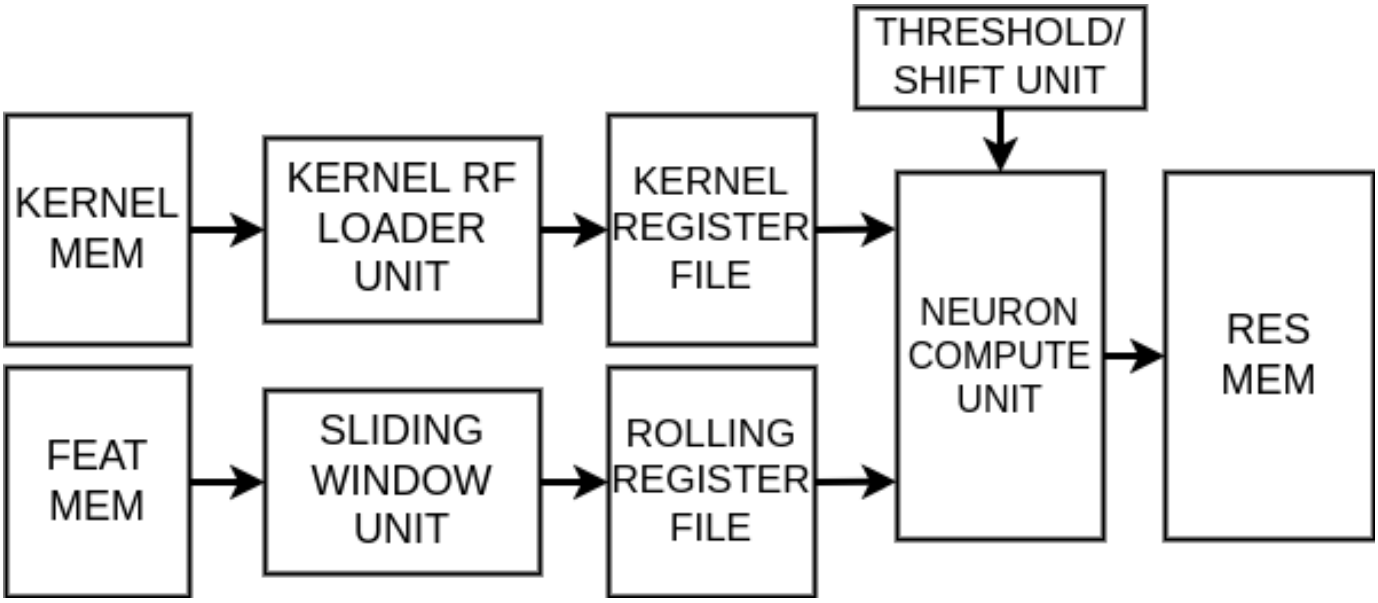

Fig. 2. A block diagram of the processing unit with arrows indicating the direction of the data flow.

```
class NeuronComputeUnit[I<:Bits, W<:Bits, M<:Bits, A<:Bits, O<:Bits](
        genI:I, genW:W, genA:A, genO:O,
        numSynaps: Int,
        mul: (I, W) => M,
        add: Vec[M] => A,
        actFn: (A, A) => O) extends Module {

    val io = IO(new Bundle {
        val in:       Vec[I] = Input(Vec(numSynaps, genI))
        val weights:  Vec[W] = Input(Vec(numSynaps, genW))
        val thresh:   A      = Input(genA)
        val shift:    UInt   = Input(UInt(8.W))
        val out:      O      = Output(genO)
    })

    val muls = VecInit((io.in.zip(io.weights)).map {
        case (x: I, w: W) => mul(x, w)
    })
    val pAct = add(muls)
    val sAct = (pAct >> io.shift).asTypeOf(pAct)
    io.out := actFn(sAct, io.thresh)
}
```

Listing 3. An abridged version of the NeuronComputeUnit module.

and the adder function as an argument, meaning it can be easily adjusted to different types of quantization, e.g. binarized neural networks. These are networks with binary weights and inputs, use the XNOR operation for multiplication, and the population count operation for the addition [2].

The unit is capable of calculating one element of the output y in a single cycle. This means that it can compute the dot product between two sets of $C * F * F$ elements in a single cycle. The amount of logic required to implement this unit is not immense, as the operational elements themselves, are low bitwidth, and fixed-point. In other words, as the use case in mind are highly quantized neural networks, we assume that the weights w bitwidth, and the input x bitwidth, are small.

D. Memory layout

One important assumption of our processing unit, is that the weights and the input feature map, can be stored on the same chip (e.g. Block RAMs on FPGAs). This means that we must be conservative in the use of memory, to allow a reasonably sized neural network to compute. Fig. 3 shows an example memory layout of an input feature map with width and height equal to three, and two channels. Each channel is represented in a different hue in the figure. Several parameters are packed together into 32-bit memory words, with no parameter being split. This constrain means that depending on the parameter bitwidth, we may have unused bits at the end of the memory word. In Fig. 3 for example, the last two bits of every word are unused.

The layout of the kernel weight parameters is similar to the layout of the input feature map parameters with the exception that the parameters of each subsequent kernel start on a new memory word. This simplifies the design with minimal memory overhead.

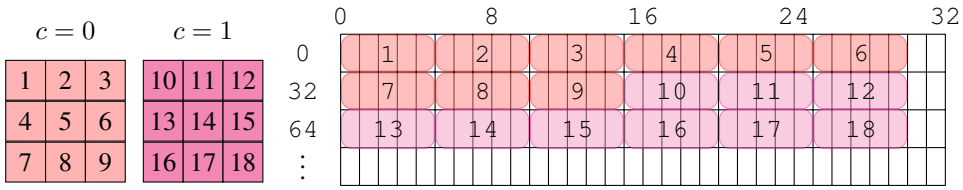

Fig. 3. Example memory layout for an input feature map quantized to five bits (C=2, W=H=3).

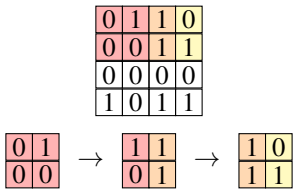

Fig. 4. The Rolling Register File operation.

E. Register Files

We use the weight stationary approach to computing the convolution. In doing so we first load the Kernel Register File with a kernel, and they then remain stationary, for the duration of sliding across the input image.

The Rolling Register File shifts feature values between registers to emulate the behavior of the window sliding across the input feature map. This behavior is demonstrated in Fig. 4, on the example input map from Fig. 1. Such a design achieves minimal data movement between the FEATMEM memory and the Rolling Register File, and also accomplishes better utilization of the Neuron Compute Unit, as we don't need to refill the entire register file from scratch every time the window moves.

F. Sliding Window Unit

The Sliding Window Unit (SWU) is responsible for gathering the feature values stored in the FEATMEM memory, and transferring them to the Rolling Register File. The SWU must be able to gather the data, regardless of the input feature map bitwidth. We implement this by using several counters and by calculating the bit-address of each specific element. The SWU is the main bottleneck in achieving a high utilization of the Neuron Compute Unit, as it jumps between different memory words to gather the specific elements of a given window. An alternative approach would be to unroll the input feature map, but this would also greatly increase the memory consumption.

IV. Experiments

Table I shows the synthesis results we obtained with constant parameters $W = H = 28$, $C = 3$, and $K = 16$, and different bitwidths of the weight and input feature map parameters. We used Xilinx Vivado 2022.2 to perform out-of-context synthesis on the Verilog generated by Chisel 3.5.5. The KBW column represents the bitwidth of the weights, IBW the bitwidth of the input feature map, LUT and FF represent the number of lookup tables and flip-flops used, $RAMB18$ and $RAMB36$ are the number of 18-kbit and 36-kbit Block RAM elements, and finally the clock represents the maximum achievable clock rate, with frequency being the inverse of it. None of our designs use any DSP blocks. We also plot the data points in Fig. 5 and Fig. 6 where KBW equals IBW, and transform the metrics $RAMB18$ and $RAMB36$ into a single metric $BRAM$, using the formula $BRAM = RAMB18 + 2 \cdot RAMB36$.

As is seen in Fig. 5 the amount of lookup tables, flip-flops, and Block RAMs our design uses is approximately proportional to the sum of the bitwidths for the case of highly quantized neural networks. The maximum frequency, however, drops off fairly quickly with increasing bitwidths, and then it reaches a plateau at around 25MHz, as it is shown in Fig. 6. In all cases the critical path was in the Neuron Compute Unit, which contains multipliers and an corresponding adder tree.

V. Conclusion

In this work we showed how Chisel can be used to write generators for convolution processing units. As a next step we intend to pipeline the Neuron Compute Unit in order to shorten the critical path and thus increase the maximum achievable frequency. We also intend to add support for other types of convolution, such as depth-wise convolution and depth-wise separable convolution.

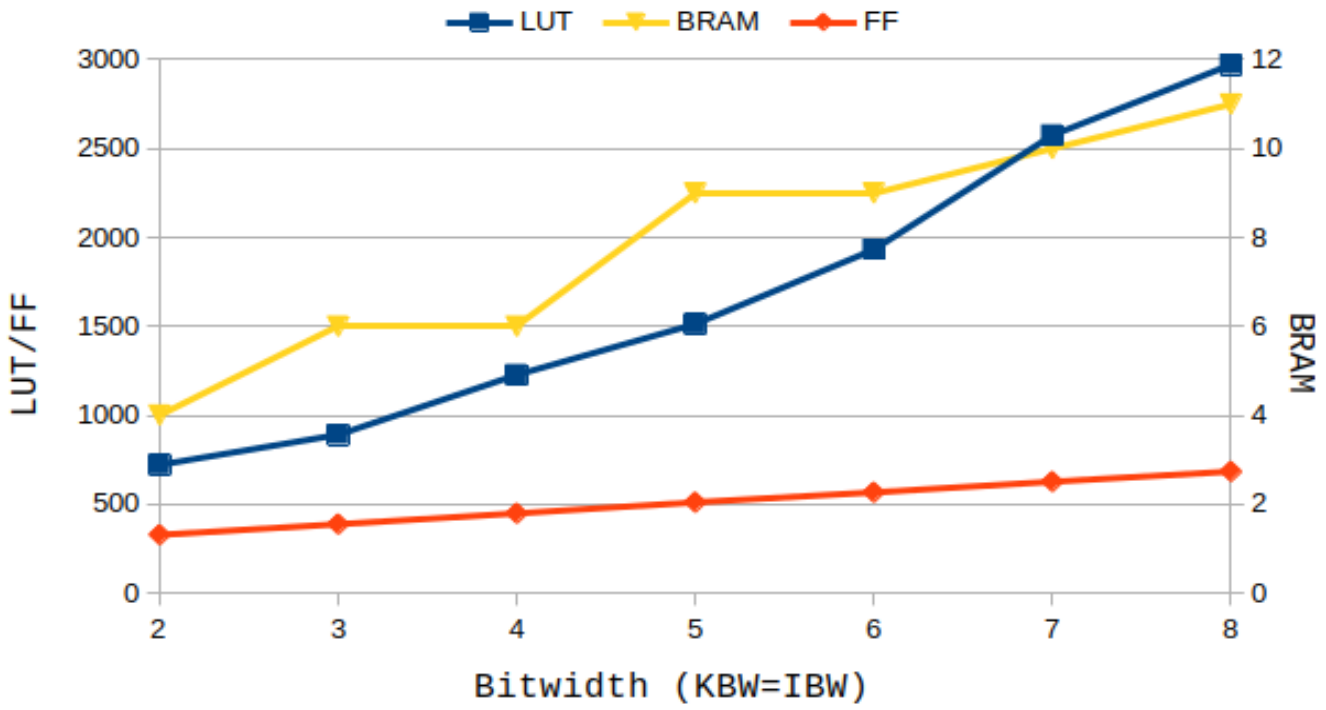

Fig. 5. A plot of the amount of resources required against the bitwidth of weights and input feature map parameters.

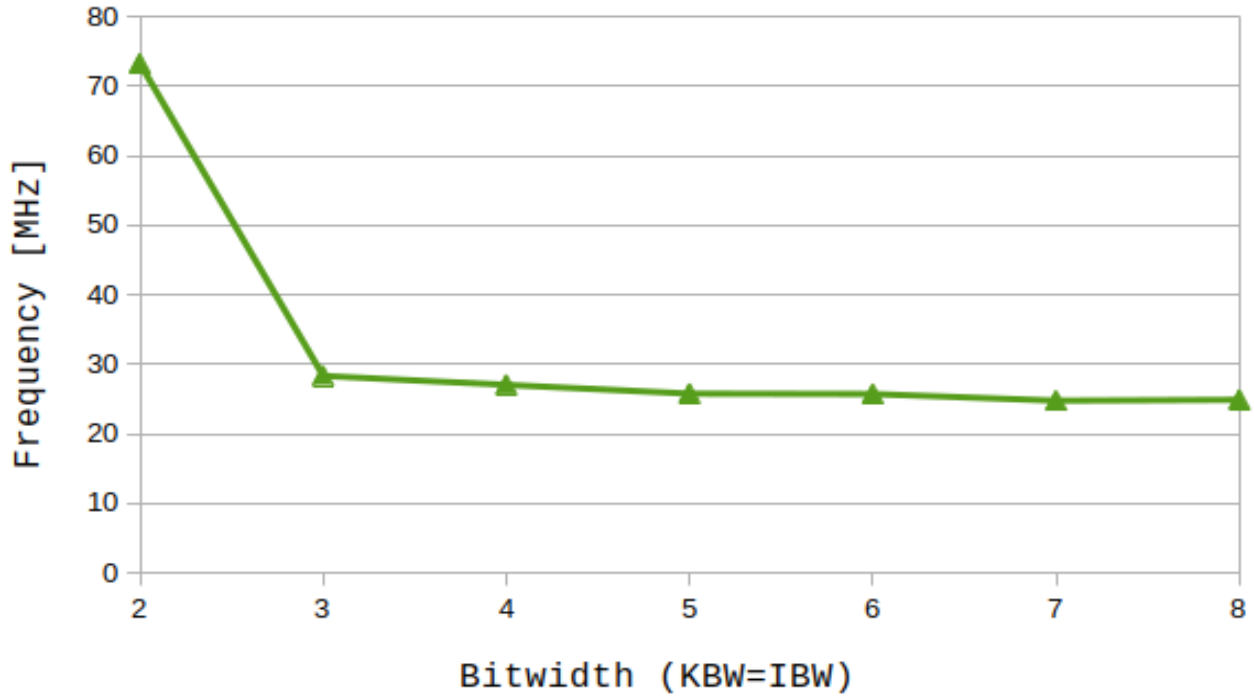

Fig. 6. A plot of the maximum achievable frequency compared to the bitwidth of weights and input feature map parameters.

References

[1] By Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Communications of the ACM* 60 (6 2012), pp. 84–90.

[2] Itay Hubara et al. "Binarized neural networks". In: *Advances in Neural Information Processing Systems* (NIPS 2016), pp. 4114–4122. ISSN: 10495258.

[3] Shuchang Zhou et al. "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients". In: 1 (1 2016), pp. 1–13. URL: http://arxiv.org/abs/1606.06160.

[4] Zhen Dong et al. "HAWQ-V2: Hessian Aware trace-Weighted Quantization of Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 18518–18529. URL: https://proceedings.neurips.cc/paper/2020/file/d77c703536718b95308130ff2e5cf9ee-Paper.pdf.

[5] Zhewei Yao et al. "HAWQ-V3: Dyadic Neural Network Quantization". In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, July 2021, pp. 11875–11886. URL: https://proceedings.mlr.press/v139/yao21a.html.

[6] Claudionor N. Coelho et al. "Ultra Low-latency, Low-area Inference Accelerators using Heterogeneous Deep Quantization with QKeras and hls4ml". In: *Nature Machine Intelligence* 3 (8 2021), pp. 675–686. ISSN: 25225839. DOI: 10.1038/s42256-021-00356-5.

[7] Michaela Blott et al. "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks". In: *arXiv* 1 (1 2018). ISSN: 23318422.

[8] Syed Asad Alam et al. "On the RTL Implementation of FINN Matrix Vector Unit". In: *ACM Trans. Embed. Comput. Syst.* (July 2022). Just Accepted. ISSN: 1539-9087. DOI: 10.1145/3547141. URL: https://doi.org/10.1145/3547141.

[9] *Chisel/FIRRTL Hardware Compiler Framework*. https://www.chisel-lang.org. Accessed: 2023-01-12.

[10] Amir Gholami et al. "A Survey of Quantization Methods for Efficient Neural Network Inference". In: (2021). URL: http://arxiv.org/abs/2103.13630.

TABLE I
SYNTHESIS RESULTS FOR $W = H = 28$, $C = 3$, $K = 16$.

KBW	IBW	LUT	FF	RAMB18	RAMB36	Clock [ns]	Freq [MHz]
2	2	721	327	2	1	13.656	73.23
2	3	911	359	2	2	14.908	67.08
2	4	692	391	2	2	34.486	29.00
2	5	825	425	3	3	34.849	28.70
2	6	949	453	3	3	36.290	27.56
2	7	1043	486	2	4	36.340	27.52
2	8	1026	515	1	5	36.017	27.76
3	2	835	355	2	1	15.157	65.98
3	3	890	387	2	2	35.289	28.34
3	4	902	419	2	2	35.282	28.34
3	5	1154	453	3	3	36.031	27.75
3	6	1279	481	3	3	36.246	27.59
3	7	1419	514	2	4	36.507	27.39
3	8	1474	543	1	5	37.251	26.84
4	2	678	384	2	1	34.495	28.99
4	3	1048	416	2	2	35.858	27.89
4	4	1227	448	2	2	36.958	27.06
4	5	1314	482	3	3	38.271	26.13
4	6	1456	510	3	3	38.081	26.26
4	7	1631	543	2	4	38.320	26.10
4	8	1701	572	1	5	38.885	25.72
5	2	708	412	2	1	34.621	28.88
5	3	1140	444	2	2	37.414	26.73
5	4	1218	476	2	2	37.305	26.81
5	5	1513	510	3	3	38.760	25.80
5	6	1791	538	3	3	38.859	25.73
5	7	2003	571	2	4	39.495	25.32
5	8	2099	600	1	5	40.057	24.96
6	2	860	440	2	1	36.554	27.36
6	3	1245	472	2	2	37.502	26.67
6	4	1342	504	2	2	37.785	26.47
6	5	1677	538	3	3	38.661	25.87
6	6	1932	566	3	3	38.851	25.74
6	7	2235	599	2	4	38.961	25.67
6	8	2386	628	1	5	39.302	25.44
7	2	912	467	2	1	36.895	27.10
7	3	1349	499	2	2	37.713	26.52
7	4	1472	531	2	2	37.926	26.37
7	5	1837	565	3	3	29.050	34.42
7	6	2150	593	3	3	38.953	25.67
7	7	2575	626	2	4	40.340	24.79
7	8	2795	655	1	5	40.100	24.94
8	2	966	495	2	1	36.616	27.31
8	3	1488	527	2	2	38.431	26.02
8	4	1617	559	2	2	38.865	25.73
8	5	1994	593	3	3	38.952	25.67
8	6	2362	621	3	3	39.194	25.51
8	7	2822	654	2	4	40.707	24.57
8	8	2972	683	1	5	40.141	24.91