Using NI LabVIEW FPGA IP Builder to Optimize and Port VIs for Use on FPGAS

Publish Date: Aug 10, 2014

Overview

FPGA IP Builder is a feature of LabVIEW that you can use to create highly optimized FPGA implementations using natural, high-level code. This white paper covers the underlying technology of FPGA IP Builder, its key benefits and examples to show how you can iteratively optimize your code for the FPGA. This paper assumes a basic understanding of FPGAs, LabVIEW, and the LabVIEW FPGA module.

Table of Contents

1. What is FPGA IP Builder?
2. How to use FPGA IP Builder
3. Directive-Driven Optimization
4. Conclusion
5. Next Steps

1. What is FPGA IP Builder?

The LabVIEW FPGA Module makes it easy to extend your LabVIEW programming skills to the FPGA. However, to create really high performance FPGA designs, you need to use the Single Cycle Timed-Loop. Because the code inside a Single Cycle Timed-Loop executes in one clock cycle, it restricts the programing elements that you can use inside it. Frequently, you have to refactor or rewrite your algorithm code using advanced techniques, like loop unrolling and pipelining to make it suitable for execution inside a Single Cycle Timed-Loop. Refactoring your code requires knowledge of your algorithm and the underlying FPGA hardware. For complex algorithms, this process can be tedious and error-prone. It can also make the resultant code harder to debug and maintain. FPGA IP Builder is a feature of the LabVIEW FPGA Module that enables you to create reusable algorithm code that you do not have to optimize manually. FPGA IP Builder enables you to:

1. Automatically optimize algorithm code for your FPGA

With FPGA IP Builder, you can create your algorithm code using high level programming elements, like arrays and loops, and automatically optimize it for your FPGA using directives. Directives are specifications that you can use to tailor the optimization process.

2. Create reusable algorithm code

Timing performance of your algorithm code is typically measured in terms of its throughput and latency. On FPGAs, timing performance is typically obtained by parallelizing your code, which tends to use more resources. Consequently, optimizing your algorithm code is a tradeoff between timing performance and resource utilization. Your optimization goals are typically governed by the requirements of your overall application.

FPGA IP Builder enables you to reuse the same high level algorithm VI in multiple applications by simply using different directives to optimize it.

3. Quickly estimate timing performance without compiling your code

For your algorithm code and a specified set of directives, FPGA IP Builder can give you an estimate of timing performance and resource utilization within minutes. Without FPGA IP Builder, these results are only available after a full compilation run which can take several minutes to an hour. The quick estimation tool enables you to rapidly iterate over many different values of directives until you meet your optimization goals.

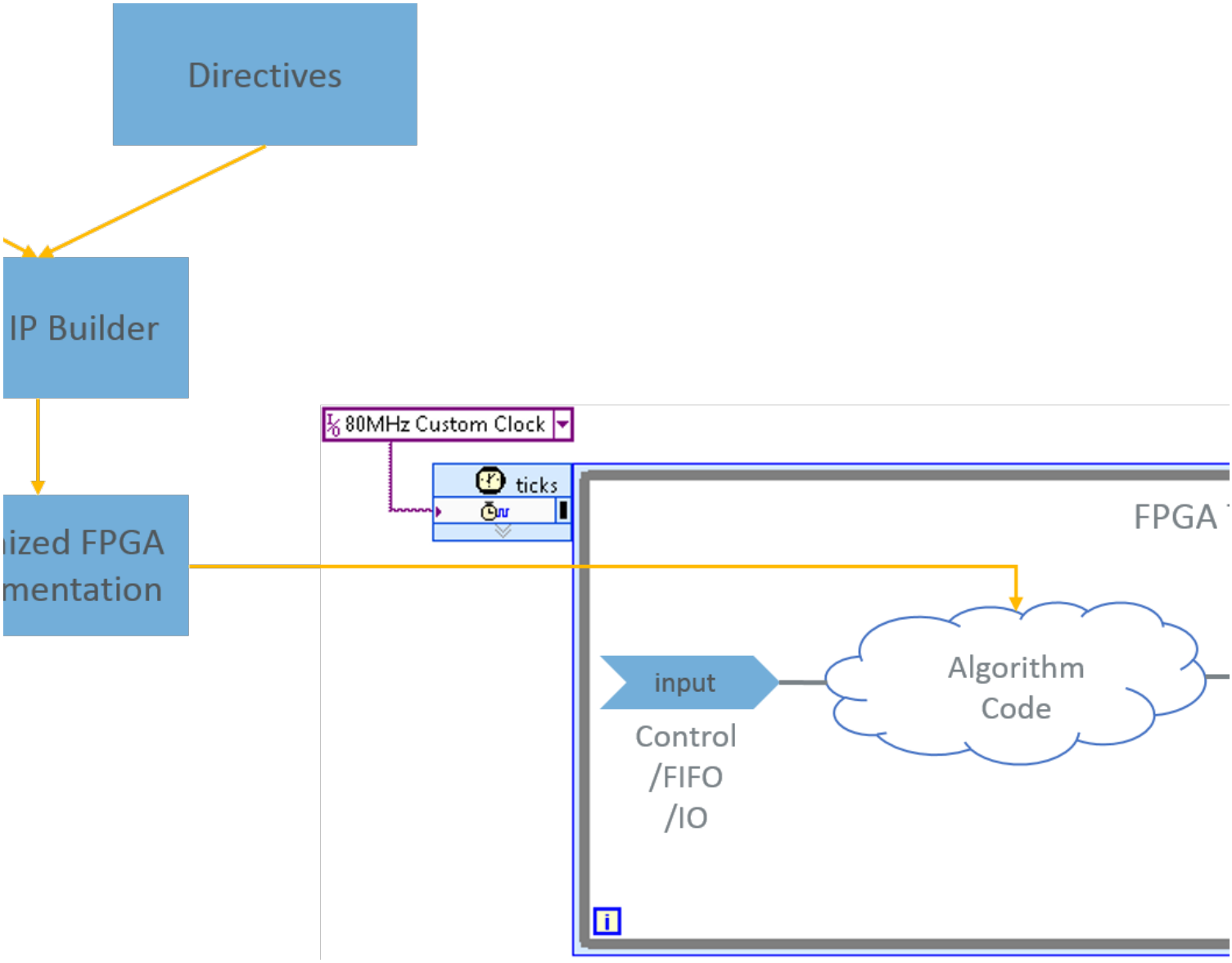


Fig 1. FPGA IP Builder uses your directives to generate an optimized FPGA implementation of your algorithm code that can be used inside a Single Cycle Timed-Loop in your Top-Level FPGA VI

2. How to use FPGA IP Builder

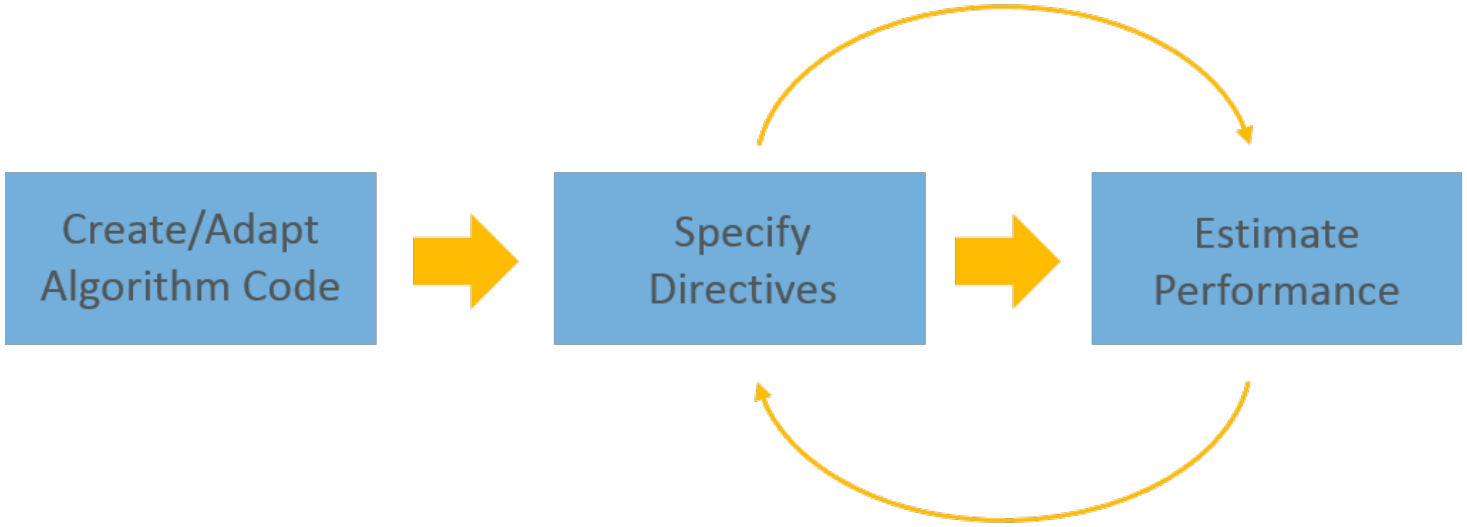
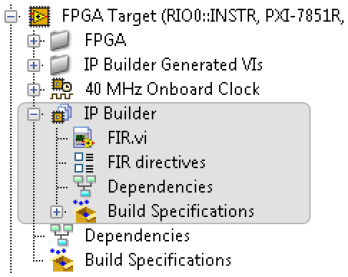


Fig 2. Typical FPGA IP Builder workflow involves taking high-level algorithm code through multiple rounds of directive driven estimation. Once your optimization goals are met, you can generate optimized code and integrate it into your Top-Level FPGA VI.

IP Builder is available as a project item under supported FPGA devices in the LabVIEW project.

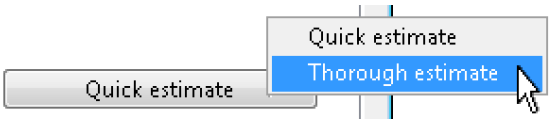


1. Create your high level algorithm VI under the IP Builder project item. Alternately, you can also copy or move VIs from other targets.
While FPGA IP Builder VIs have fewer palette restrictions than Single Cycle Timed-Loops, they do not support all functions and data types as LabVIEW on Desktop. Refer to product documentation for details on supported functions and data types.

Algorithm VIs should be hardware agnostic and should not contain IO, timing or hard-coded constants.

2. Create a directives item for your algorithm VI. You can specify your directives and run quick estimations in the directives property dialog. Choosing directives and evaluating results is covered in the next section. You can create one or more sets of directives for your algorithm VI.

3. Once you achieve your optimization goals in the estimation tool, you should run a thorough estimation to validate your design. Thorough estimation provides a very accurate performance and resource estimate by actually compiling your optimized algorithm code using the Xilinx toolchain.



4. Create a Build Specification from your directives to generate the optimized FPGA implementation of your algorithm VI. The Build Specification generates a VI containing the optimized FPGA implementation in a folder named IP Builder generated VIs under your FPGA target.

5. Create a top-level VI under your FPGA target containing a Single Cycle Timed-Loop with the necessary timing and IO. Insert the FPGA IP Builder generated VI into the Single Cycle Timed-Loop.

3. Directive-Driven Optimization

The following section explains the use of directives to optimize your algorithm code using example. Each example lists the location of the source VI and a step by step procedure to achieve the stated optimization goal. For a detailed procedure, refer to the FPGA IP Builder Tutorial in the Product Documentation.

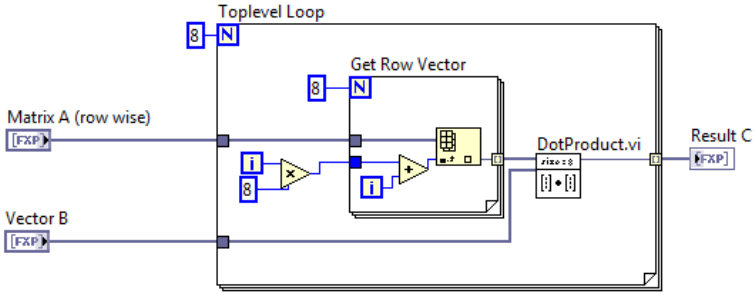
NOTE: the examples below are run on a NI Compact RIO 9068 target. Results presented here might not replicate exactly. If you see significant discrepancies while replicating these results, please report it on the LabVIEW FPGA IP Builder Community.

Optimization Example 1: Matrix Vector Multiplication

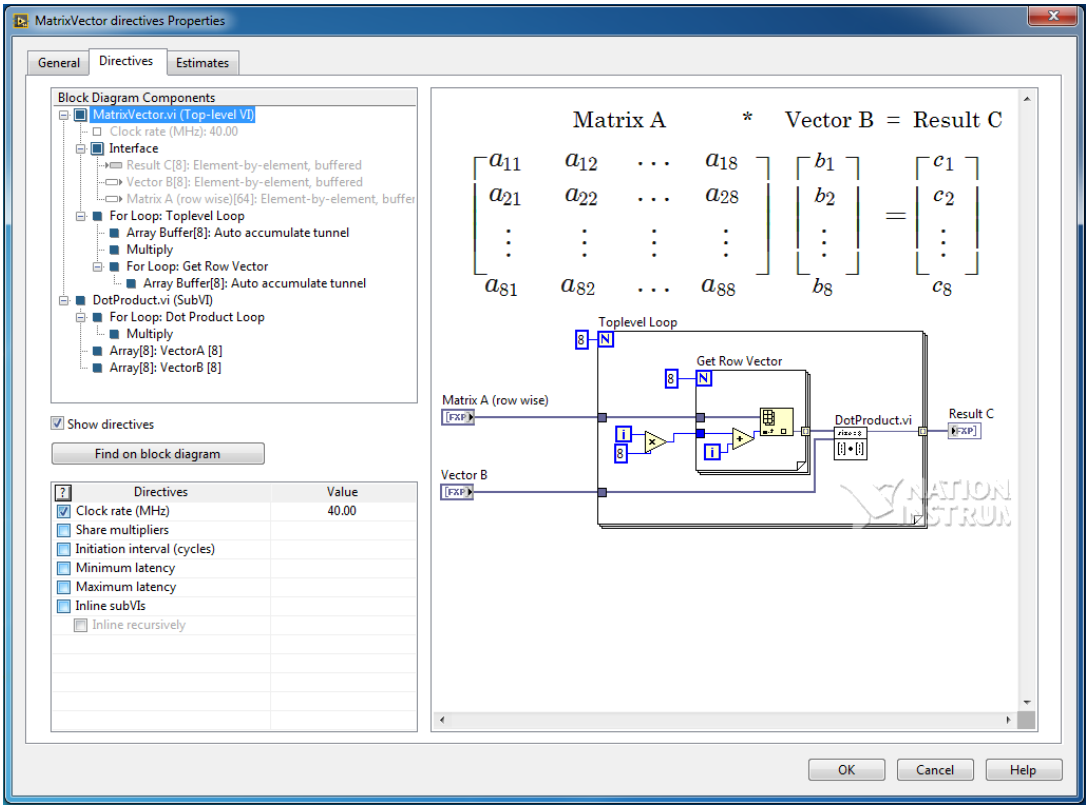
Objective: This example demonstrates the use of the Initiation Interval directive to maximize Throughput.

Source VI: Consider the matrix vector multiplier code below. You can download this example at the LabVIEW FPGA IP Builder Community.

$$\underset{\text{Matrix A}}{\begin{bmatrix} a_{11} & a_{12} & \dots & a_{18} \\ a_{21} & a_{22} & \dots & a_{28} \\ \vdots & \vdots & \vdots & \vdots \\ a_{81} & a_{82} & \dots & a_{88} \end{bmatrix}} * \underset{\text{Vector B}}{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_8 \end{bmatrix}} = \underset{\text{Result C}}{\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_8 \end{bmatrix}}$$



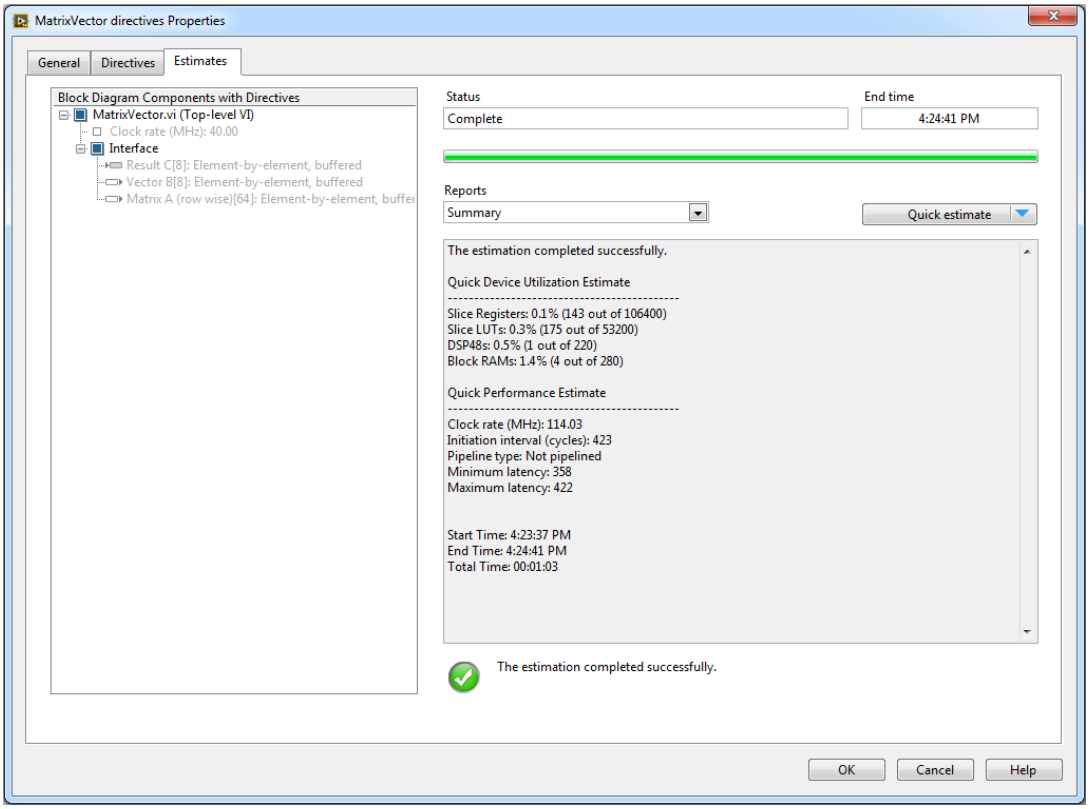
Setup: Copy the source VI under the IP Builder project item. Right-click MatrixVector.vi and select **create Directives** from the shortcut menu. LabVIEW creates an **MatrixVector directives** project item above MatrixVector.vi. Double-click **MatrixVector directives** to display the MatirxVector directives Properties dialog box. Navigate to the **Directives** tab.



The **Directives** tab shows you the Block Diagram of MatrixVector.vi. In the top-left corner, you can see a hierarchical list of Block Diagram components. The bottom-left corner shows directives that you can configure for a selected component in the Block Diagram hierarchy. Directives that are specified with the Top-Level VI selected are referred to as **Top Level Directives** in this paper.

Iteration 1

The only directive set by default is **Clock rate (MHz)** at 40 MHz. This is the default base clock on the FPGA. Run a quick estimate with the default directive to establish the baseline timing performance of your design. Navigate to the **Estimates** tab and click **Quick estimate** to generate an estimation report.



The **Report Summary** contains a Device Utilization estimate that lists the percentage of FPGA hardware resources used by your design. A thorough understanding of FPGA hardware resources is not necessary to follow this example. To learn more about them, refer to product documentation here and here.

The **Report Summary** also contains a Performance estimate in terms of Clock rate (MHz), Initiation interval (cycles) and minimum and maximum latency.

Clock rate (MHz) is the frequency of the clock that drives the Single Cycle Timed-Loop containing your algorithm code. If your Single Cycle Timed-Loop can process one sample of data per iteration, a faster clock rate directly improves performance. Clock rate is ultimately limited by the critical path in your design. Critical path is the sequence of logical/mathematical operations that takes the longest time to execute in your design. Specifying a high clock rate causes FPGA IP Builder to pipeline your design and break up the critical path whenever possible. To learn more about pipelining, refer to product documentation here.

Initiation Interval (cycles) is the number of cycles between inputs to your design. Its basically a measure of how frequently your design can accept new inputs. On FPGA hardware, inputs are typically passed into logic blocks using clock pulses or ticks. If your design can accept an input on every consecutive tick, there is a one clock period (or cycle) between them. This implies an initiation interval of 1. Setting this directive causes FPGA IP Builder to recursively unroll loops until your Initiation Interval specification can be met.

Latency is the number of clock cycles that it takes your design to complete one iteration of your algorithm VI. It can also be interpreted as the time between an input entering your design and a corresponding output being generated. In certain applications, like closed loop control, this time is an important consideration.

Throughput is a key metric of performance for your algorithm but it is not directly estimated by the estimation tool. Throughput is the number of samples of data that your design can process per unit time. You can calculate throughput for your design using the formula below

$$Throughput = \frac{Clock\ Rate \times Samples\ per\ Iteration}{Initiation\ Interval}$$

Samples per Iteration is simply the number of samples that your design can accept per call. You can look at the inputs of your algorithm VI to learn this.

If you analyze **MatrixVector.vi**, you will see that it accepts one 8x8 matrix per iteration of the VI. You can consider this one sample to calculate throughput. Therefore, Samples per Iteration is 1. To calculate the Throughput, use Initiation Interval obtained from the quick estimate summary.

NOTE: Clock rate in the quick estimate summary might be higher than the directive specified. However, it might not always be possible to generate an arbitrary clock rate on the FPGA. Further more, your Single Cycle Timed-Loop clock rate might be limited by other functionality in the loop. Hence this paper uses the clock rate specified in the directives to calculate throughput.

$$Throughput = \frac{40\ MHz \times 1\ Sample}{282\ cycles} = 0.14\ MS/s$$

Throughput of your design is measured in megasamples per second.

The table below summarizes the inputs and outputs of the quick estimation

Top Level Directives: Clock Rate (MHz) : 40 MHz

	Slice Registers (%)	Slice LUTs (%)	DSP48s (%)	BRAMs (%)	Clock Rate Input (MHz)	Initiation Interval (cycles)	Min. Latency (cycles)	Max. Latency (cycles)	Throughput (MS/s)
Iter. 1	1.6%	4.9%	0.5%	0.4%	40	282	281	281	0.14

Iteration 2

From the formula above, it is clear that throughput scales directly with the clock rate. Therefore increasing clock rate should get you higher throughput. Set **Clock rate (MHz)** to 200 MHz and run a quick estimate.

NOTE: Your choice of clock rate is typically governed by your overall design - specifically other other logic in your Single Cycle Timed-Loop. If you merely need to find out the fastest possible clock rate that you drive your logic with, you can keep increasing the value of the **Clock rate (MHz)** until the quick estimation tool reports a failure to meet it.

The table below summarizes the inputs and outputs of the quick estimation. Note that Throughput increases to 0.41 MS/s which is a 2.9x increase over the baseline.

.

Top Level Directives: Clock Rate (MHz) : 200 MHz

	Slice Registers (%)	Slice LUTs (%)	DSP48s (%)	BRAMs (%)	Clock Rate Input (MHz)	Initiation Interval (cycles)	Min. Latency (cycles)	Max. Latency (cycles)	Throughput (MS/s)
Iter. 1	1.6%	4.9%	0.5%	0.4%	40	282	281	281	0.14
Iter. 2	**1.8%**	**4.9%**	**0.5%**	**0.4%**	**200**	**482**	**481**	**481**	**0.41**

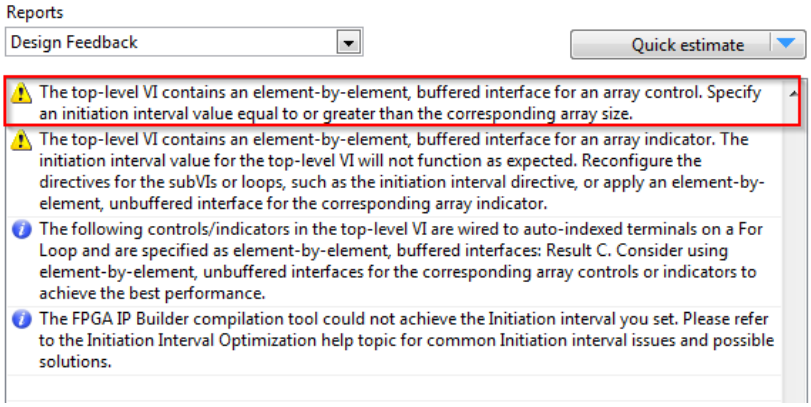
Iteration 3

Based on the formula above, throughput is inversely proportional to initiation interval. Therefore, reducing Initiation interval should also give higher throughput. Keep clock rate at 200MHz and set **Iteration Interval** to 1. Run a quick estimate and review the results.

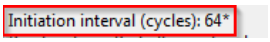
Note that estimation tool reports that it is not possible to achieve an initiation interval of 1

The estimation completed successfully. However, some directives cannot meet the performance requirements you configured.
* Directives that cannot meet the requirements are shown in the Quick Performance Estimate report in bold and with an asterisk.

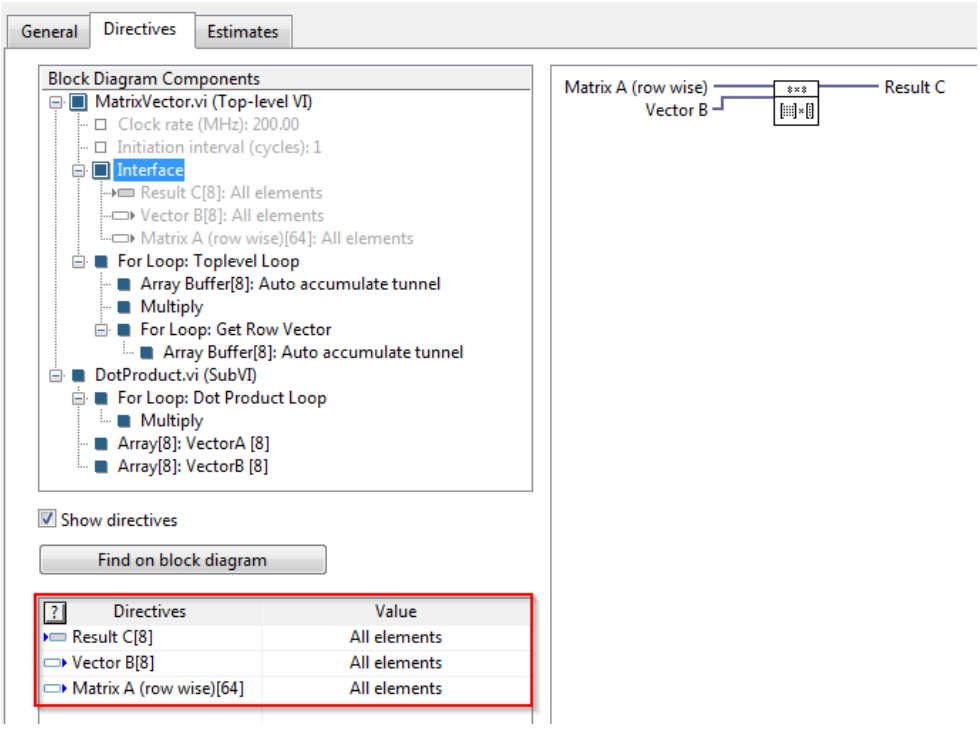
You can review the **Design Feedback** from the estimation tool for suggestions on fixing the issue. **Design Feedback** is available as a report from the estimation tool



When you have IP with arrays as inputs and outputs, IP Builder automatically configures them to be streaming interfaces. When an interface is set to streaming, instead of expecting all elements of the array to appear at the input (or output) at once, they appear element-by-element - one element per clock cycle. This means that, with the input array configured as an element-by-element interface, your IP cannot have an initiation interval of shorter than the number of elements in the input array. With your IP, the shortest initiation interval can be 64. The estimation report reflects this



The selection of receiving your input array as all elements at once or receiving them element-by-element depends on the rest of your design, specifically how preceding logic outputs data. To maximize throughput, you can set your interfaces to **All elements** by selecting them from the **Block Diagram Components** hierarchy



The table below shows inputs and outputs of the quick estimation tool after setting the interface directives. Note that throughput increases to 200Ms/s - 1428x improvement over baseline. However, your design uses nearly 30% of all DSP48s on the FPGA. DSP48s are highly efficient multipliers on the FPGA. With an initiation interval of 1, all 64 elements of the input matrix are multiplied in one clock cycle. This requires 64 DSP48s in parallel.

Top Level Directives: Clock Rate (MHz) : 200 MHz, Initiation Interval (cycles) : 1, Interface: All elements

	Slice Registers (%)	Slice LUTs (%)	DSP48s (%)	BRAMs (%)	Clock Rate Input (MHz)	Initiation Interval (cycles)	Min. Latency (cycles)	Max. Latency (cycles)	Throughput (MS/s)
Iter. 1	1.6%	4.9%	0.5%	0.4%	40	282	281	281	0.14
Iter. 2	1.8%	4.9%	0.5%	0.4%	200	482	481	481	0.41
Iter. 3	**3.5%**	**1.6%**	**29.1%**	**0%**	**200**	**1**	**5**	**5**	**200.00**

Iteration 4

Setting initiation interval to 1 gives you a significant improvement in throughout at the cost of nearly 30% of DSP48s on your system. If you need to implement multiple matrix vector multipliers on your FPGA, the dependency on DSP48s would restrict you to only 3 even if you did not max out other resources on the FPGA. To explore the performance/resource tradeoff, you can set the initiation interval to 2 and run a quick estimate. Note that this reduces your throughput by half (as expected) but also reduces the DSP48 utilization. With nearly 15% utilization of DSP48s, you can now fit 6 matrix vector multipliers on your FPGA.

The table below shows inputs and outputs of the quick estimation.

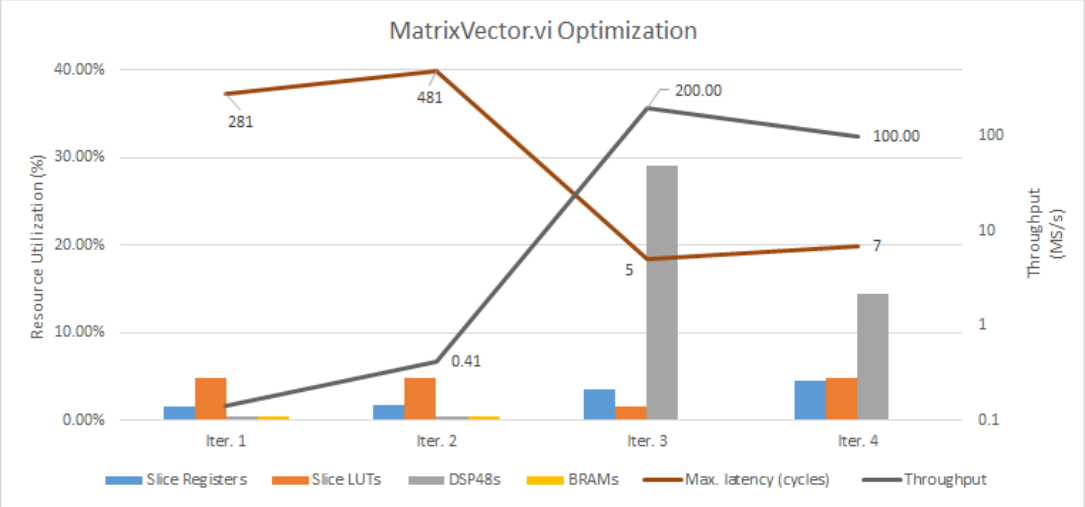
Top Level Directives: Clock Rate (MHz) : 200 MHz, Initiation Interval (cycles) : 2

	Slice Registers (%)	Slice LUTs (%)	DSP48s (%)	BRAMs (%)	Clock Rate Input (MHz)	Initiation Interval (cycles)	Min. Latency (cycles)	Max. Latency (cycles)	Throughput (MS/s)
Iter. 1	1.6%	4.9%	0.5%	0.4%	40	282	281	281	0.14
Iter. 2	1.8%	4.9%	0.5%	0.4%	200	482	481	481	0.41
Iter. 3	3.5%	1.6%	29.1%	0%	200	1	5	5	200.00
Iter. 4	**4.5%**	**4.9%**	**14.5%**	**0%**	**200**	**2**	**7**	**7**	**100.00**

Takeaway

The **Iteration Interval** directive forces FPGA IP Builder to unroll loops until the design can accept one sample per cycle. Unrolling loops leads to a highly parallel design that consumes more resources on the FPGA.

Each quick estimation run takes under a minute on average. Using directive-driven optimization, within minutes, you can achieve nearly 1500x improvement in throughput and nearly 1/5th the latency of the original design. More importantly, you can also use it to explore design tradeoffs and pick the right optimization to meet your application goals.



Understanding the significance and relative priority of directives can enable you to drive to your desired result faster. Refer to the product documentation to better understand directives.

Optimization Example 2

Objective: When your IP has array inputs or outputs, FPGA IP Builder sets them as element-by-element interfaces by default. The choice of how to configure your interfaces really depends on how logic upstream and downstream processes the data. Example 1 used the "All Elements" interface type to maximize throughout. Example 2 demonstrates the use of the element-by-element interface and Share Multipliers directive to reduce resource usage while maintaining the highest throughput possible.

Source VI: Example 2 uses the same VI as Example 1

Setup: Open the source project and create a directives project item for the source VI.

Iteration 1

As seen in example 1, with the default element-by-element interface, the lowest initiation interval that can be achieved is 64. The baseline throughout for example is calculated by the formula below

$$Throughput = \frac{200\ MHz \times 1\ Sample}{64\ cycles} = 3.13\ MS/s$$

NOTE: Throughput can be further increased by choosing a higher clock rate. However your choice of clock rate is typically governed by your overall design - specifically other other logic in your Single Cycle Timed-Loop. Since the goal of this example is not maximizing throughput, it uses the clock rate from example 1 as a baseline.

The following table shows the inputs and outputs of the quick estimate

Top Level Directives: Clock Rate (MHz) : 200 MHz

	Slice Registers (%)	Slice LUTs (%)	DSP48s (%)	BRAMs (%)	Clock Rate (MHz)	Initiation Interval (cycles)	Min. Latency (cycles)	Max. Latency (cycles)	Throughput (MS/s)
Iter. 1	3.6%	0.8%	3.6%	0%	200	64	136	136	3.13

Iteration 2

You have two choices for configuring inputs as element-by-element types. If incoming array elements are accessed in the sequence they come in and are used only once, then they do not need to be saved using buffers in your IP. If incoming elements are accessed out of order, then they need to be buffered inside your IP. You can choose to configure your element-by-element interfaces as buffered or unbuffered by looking at the access patterns of your IP. In the MatrixVector example, elements of array **Matrix A (row-wise)** are only used once each in the matrix vector computation. Elements of Array **Vector B** on the other hand are used thrice in the computation. Therefore, **Matrix A (row-wise)** can be set to **Element-by-element, Unbuffered. Vector B** cannot be set to Unbuffered. Outputs do not need non sequential or multiple access so they can be set to Unbuffered.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = a_{11} \times b_1 + a_{12} \times b_2 + a_{13} \times b_3 + a_{21} \times b_1 + a_{22} \times b_2 + a_{23} \times b_3 \ldots$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{21} & a_{22} & a_{23} & \ldots & a_{33} \end{bmatrix}$$

$$\begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}$$

The following table shows the inputs and outputs of the quick estimate with **Matrix A (row-wise)** and **Result C** set to **Element-by-element, Unbuffered.** Note that the number of Slice Registers reduces from Iteration 1.

Top Level Directives: Clock Rate (MHz) : 200 MHz, Initiation Interval : 64

Interface Directives: Matrix A (row wise), Result C: Element-by-element, unbuffered

	Slice Registers (%)	Slice LUTs (%)	DSP48s (%)	BRAMs (%)	Clock Rate (MHz)	Initiation Interval (cycles)	Min. Latency (cycles)	Max. Latency (cycles)	Throughput (MS/s)
Iter. 1	3.6%	0.8%	3.6%	0%	200	64	136	136	3.13
Iter. 2	**1.1%**	**0.8%**	**3.6%**	**0.0%**	**200**	**64**	**73**	**73**	**3.13**

Iteration 3

The **Share Multipliers** directive guides IP Builder to share DSP48s wherever possible. Set **Share Multipliers** to true and run a quick estimate. Note that number of DSP48s required goes from 8 to 2.

Top Level Directives: Clock Rate (MHz) : 200 MHz, Initiation Interval : 64, Share Multipliers : True

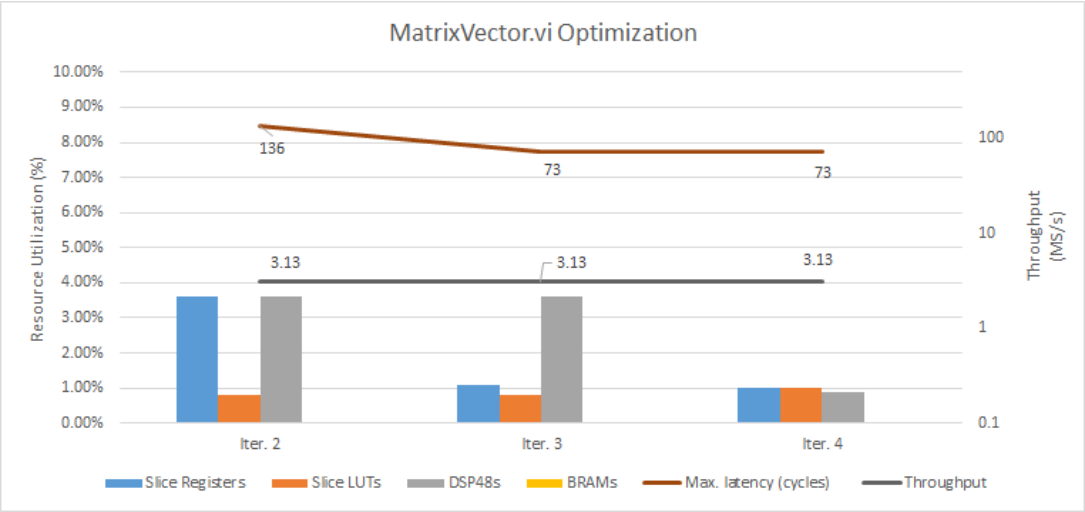
Interface Directives: Matrix A (row wise), Result C: Element-by-element, unbuffered

	Slice Registers (%)	Slice LUTs (%)	DSP48s (%)	BRAMs (%)	Clock Rate (MHz)	Initiation Interval (cycles)	Min. Latency (cycles)	Max. Latency (cycles)	Throughput (MS/s)
Iter. 1	3.6%	0.8%	3.6%	0%	200	64	136	136	3.13
Iter. 2	1.1%	0.8%	3.6%	0.0%	200	64	73	73	3.13
Iter. 3	**1.0%**	**1.0%**	**0.9%**	**0.0%**	**200**	**64**	**73**	**73**	**3.13**

Takeaway

You can use directive-driven optimization to easily convert your design to accept streaming inputs without modifying code. For certain designs, you can use Share Multipliers and interface directives to reduce resource utilization while still retaining gains in throughout and reduction in latency.

Setting the Element-by-element, unbuffered directive wherever possible can lead to saving in registers and associated circuitry. This can be especially significant if the arrays are large. Refer to the product documentation (see problem #3 and possible solutions) for more tips on improving FPGA resource utilization.



4. Conclusion

You can use FPGA IP Builder to automatically optimize your high level algorithm VIs for your FPGA. As evidenced by the examples, within a few iterations, you can obtain almost 1500x increase in throughput without code modifications. The quick estimation process typically takes under a minute per iteration and enables you to iteratively optimize your code within minutes.

5. Next Steps

Starting in 2014, FPGA IP Builder is a feature of the LabVIEW FPGA Module. The principles outlined in this paper can help you get started with algorithm optimization. In many cases, you can extract an even higher performance by starting with well designed code than you can by only using directives. For code creation best practices, training and examples, visit the FPGA IP Builder Community. You can also directly communicate with FPGA IP Builder developers on the community by asking them specific questions.