FPGA Acceleration of Matrix Multiplication for Neural Networks

XAPP1332 (v1.0) February 27, 2020

Summary

This application note describes the implementation and evaluation of a large multiply-add systolic array designed for the acceleration of matrix multiplication for deep learning neural network inference applications. This design is built on an array of 6144 DSPs in a *32×192* configuration, spanning all 3 super logic regions (SLRs) of the XCVU37P-2E FPGA. Timing closure was achieved with a maximum operating frequency of 638 MHz.

Download the reference design files for this application note from the Xilinx® website. For detailed information about the design files, see Reference Design.

Introduction

Matrix Multiplication in Neural Networks

Matrix multiplication is the most demanding operation in deep learning inference in terms of computational resources. The number of multiply-add operations total many billions in modern neural networks. Therefore, a competitive inference system requires a fast and efficient matrix multiplier as the main computational engine. High throughput convolutional matrix multiplication with systolic multiply-add arrays on FPGAs has been previously demonstrated at the maximum FPGA operating frequency, f_{MAX} [Ref 1] [Ref 2]. High computational efficiency of systolic arrays in the acceleration of an image classification neural network has been demonstrated with careful matching of the dimensions of the systolic array and neural network layers [Ref 3] [Ref 4].

Even though modern neural networks consist of a wide range of dimensions of layers, numbers of channels, and kernels for image-processing networks, the matrix dimensions tend to be quite large. Therefore, a single large systolic multiplier array, using the FPGA resources, is easily programmable and can be readily and efficiently applied to any neural network.

Matrix-Matrix Multiplication Decomposed into Matrix-Vector

The matrix multiplication problem in a given neural network layer can be written as:

Equation 1: **Matrix Multiplication**

$$Y = AB + c1$$

Here, the layer input operands are matrices *A* and *B* with dimensions *[M×K]* and *[K×N]*, respectively, a column vector *c* with dimensions *[M×1]*, and a row vector ***1*** with dimensions *[1×N]*. The resulting layer output, *Y*, has dimensions *[M×N]*. In image-processing layers, the operands *A* and *B* can be regarded as weights and activations, respectively, and the vector *c* as bias. The matrix inner product dimension, *K*, is the unrolled filter kernel, which is the product of the filter dimensions and the number of input channels. The left-hand output dimension, *M*, maps to the number of output channels. The right-hand output dimension, *N*, maps to the output image area. The layer computation in Equation 1 can be efficiently performed as a series of *N* independent matrix-vector multiplication operations of the *[M×K]* matrix *A* by a column vector *b* with length *[K]*.

Equation 1 is mapped to a hardware multiplier array that performs the operation in Equation 2 every clock cycle:

Equation 2: **Hardware Multiplier Array Operation**

$$y' = A'b' + c'$$

The hardware performs a multiplication of matrix *A'* by a column vector *b'* with dimensions *[m×k]* and *[k×1]*, respectively, with an addition of a bias column vector *c'* with dimensions *[m×k]*. So, *k* is the length of the inner product that can be performed every clock cycle, generating *m* simultaneous outputs. In the implementation, there are *m* dot-product multiply-accumulation cascade chains with length *k*. The right-hand vector operand *b'* is broadcast to all *m* cascade chains, operating in parallel. Because the matrix-vector multiplication operation is the basis of the accelerator operation, this hardware block is referenced as *M×V*.

Matrix Multiplication Mapped to *M×V*

The mapping of a matrix multiplication problem is shown in Figure 1. In this example, the matrix multiplication inner product dimension, *K*, is mapped onto the *M×V* multiply-accumulate cascade length, *k*. The following equation illustrates the minimum passes required to complete the matrix inner dot product.

$$\mathrm{P}_k = \left\lceil \frac{K}{k} \right\rceil$$

Similarly, *M*, the number of rows in matrix *A*, is mapped onto the *m* output lanes of the *M×V*. The following equation illustrates the minimum passes required to complete all *M* output rows.

$$\mathrm{P}_m = \left\lceil \frac{M}{m} \right\rceil$$

Each inner dot-product pass is labeled as:

$$\rho_k \in [0,\ \mathrm{P}_k)$$

Each pass along the output row axis is labeled as:

$$\rho_m \in [0,\ \mathrm{P}_m)$$

In general, *K* might not be an integer multiple of *k*, and *M* might not be an integer multiple of *m*. So, some zero padding of operand matrices *A* and *B* might be required. The last pass along the inner dot product axis or the output row axis with zero padding is shown in shaded gray areas at the edges of matrices *A* and *B* in Figure 1.

Inner Dot Product Axis:

$$\rho_k = \mathrm{P}_k - 1$$

Output Row Axis:

$$\rho_m = \mathrm{P}_m - 1$$

The dimensions of zero-padded matrices A_{pad} *and* B_{pad} are:

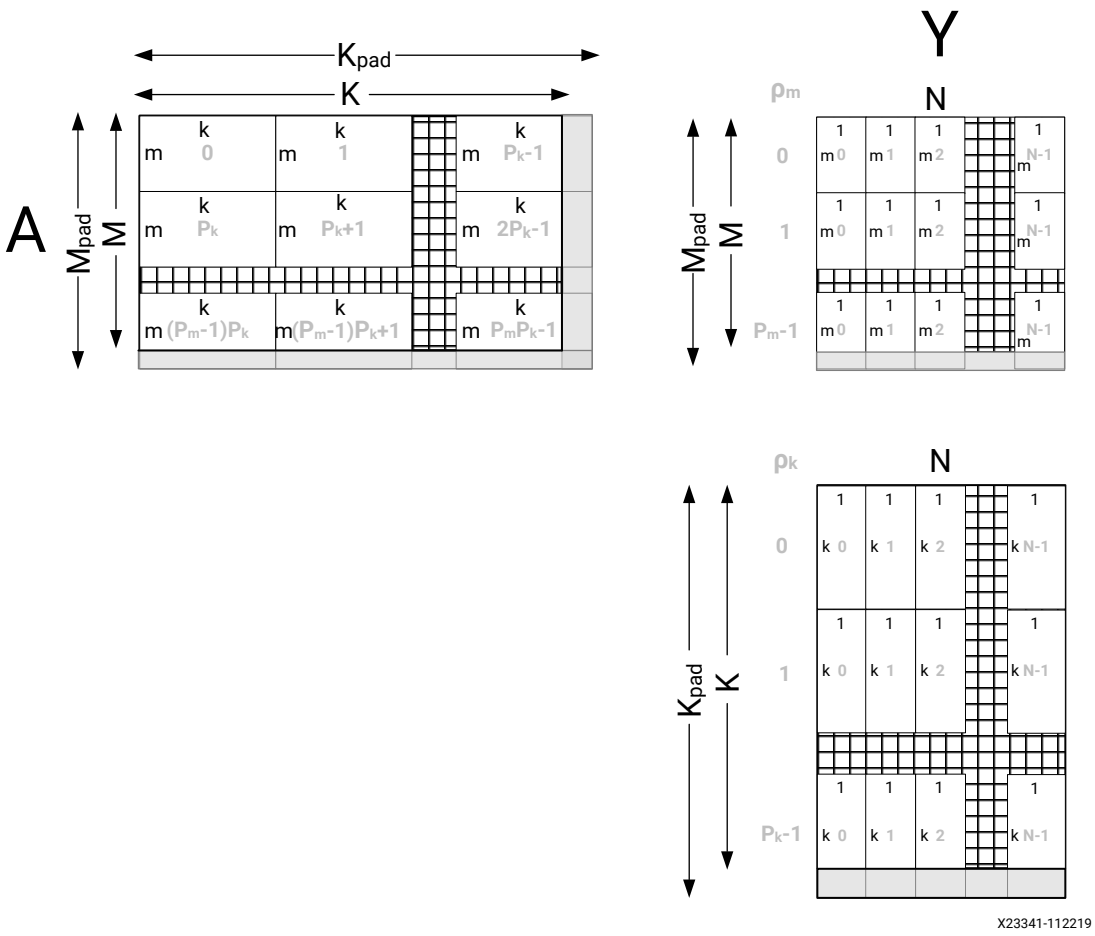
$$[\mathrm{P}_m m \times \mathrm{P}_k k]$$
$$[\mathrm{P}_k k \times N]$$

The left-hand operand of Equation 1, *A*, is divided into $\mathrm{P_m P_k}$ blocks, each with dimensions *[m×k]*. These data blocks, numbered $\mathrm{P_m P_k} - 1$ in Figure 1, are loaded one after another into the *A'* register of the *M×V*. Similarly, the right-hand operand of Equation 1, *B*, is divided into $\mathrm{P_k N}$ vectors, each with length *[k]*, which are loaded into the *b'* input of the *M×V*. For each of the ρ_m passes along the output row axis, a bias vector with length *[m]* is loaded into the *c'* input of the *M×V*.

Multiplication Scheduling to Minimize Required Memory Bandwidth

Due to limited memory bandwidth, multiple cycles are required to load the entire sub-block *A'* into the *M×V* register. Therefore, as-late-as-possible (ALAP) scheduling is used to hold the *A'* data as long as possible while loading the complementary page. This minimizes the required memory bandwidth for loading *A'*. Each of the ρ_m ρ_k *A'* sub-blocks is multiplied by *N b'* vectors corresponding to the ρ_k index. This requires the accumulation of *N* partial sums for *m* outputs corresponding to the ρ_m index. There is a maximum limit on the number of partial sums that can be kept in memory: $N \leq N_{acc}$, where N_{acc} is the maximum depth of the accumulator memory. Operand *B* matrices with $N \geq N_{acc}$ are separated into smaller sub-matrices that satisfy this constraint.

Figure 1: **Matrix Multiplication mapping onto the MxV**



Operation Description

Precisions Supported (Input and Output)

The *M×V* currently supports precisions of matrix multiplication operands listed in the following table. There are *[32×192]* physical multipliers implemented, capable of performing multiply-add operations with 16-bit operands every clock cycle. With 8-bit operands, the number of output rows can be doubled to 64, with a small overhead in the dot-product accumulation chain [Ref 5] [Ref 6]. The precision of the output result is reduced from 64 bits to 32 bits for 8-bit operands.

With 16-bit operands, 1 row of matrix *A'* is loaded every cycle. This value is labeled as m_A in the following table. For 8-bit operands, m_A *=2*, because the maximum memory bandwidth is held constant. Lastly, the number of cycles to fully populate one page of the *A'* register is the same for both operand precisions, because the following equation is used for both cases.

$$m_{phys} = \frac{m}{m_A} = 32$$

Table 1: **Supported Input Operation and Precision by the *MxV***

A type	*B* type	*Y* type	*M×V* size *[m x k]*	*A'* Rows loaded per cycle m_A
int8	int8	int32	64×192	2
uint8	int8	int32	64×192	2
int8	uint8	int32	64×192	2
uint8	uint8	uint32	64×192	2
int16	int16	int64	32×192	1
uint16	int16	int64	32×192	1
int16	uint16	int64	32×192	1
uint16	uint16	uint64	32×192	1

Scaling

Scaling is often used in neural networks to retain dynamic operating range with quantized layer operands. The dynamic range for the precisions listed in Table 1 for the matrix multiplication operands in Equation 1 can be expanded through the use of scaling factors. The *M×V* supports independent scaling control of the matrix multiplication operands along the *M*-element row axis of matrix operand *A*. So, s_A and s_c, the scaling factors for matrix multiplication operand *A* and bias vector *c*, respectively, are both vectors with length *M*. The scaling factor for matrix multiplication operand *B*, s_b, is a scalar value.

Clipping

The matrix multiplication output can be passed through a fused non-linear clipping operation. This operation, when enabled, restricts the output to be between two values, $clip_{min}$ and $clip_{max}$:

$$Y_o = \min(clip_{max}, \max(Y, clip_{min}))$$

Python User Interface

A set of user interface software tools is provided in the python 3 operating environment as part of the *M×V* package. These tools are developed to accept the *A*, *B*, and *c* matrix multiplication operands as input to the software in the form of numpy arrays [Ref 7], in one of the formats listed in Table 1. The user interface software generates the corresponding formatted binary vector inputs to the *M×V*, as well as the expected reference data vectors. A software usage example is provided in the package in `\python\MxV_genInput.py` python script. In this script, the command and data input vectors to the *M×V* are generated by the `xal.MxV_queues.gen_queue_inputs()` function after all the parameters are defined. As part of the demonstration of correctness, the reference inputs stored in SRAM in this package include the vectors generated by this example script.

Vivado Tcl Scripts

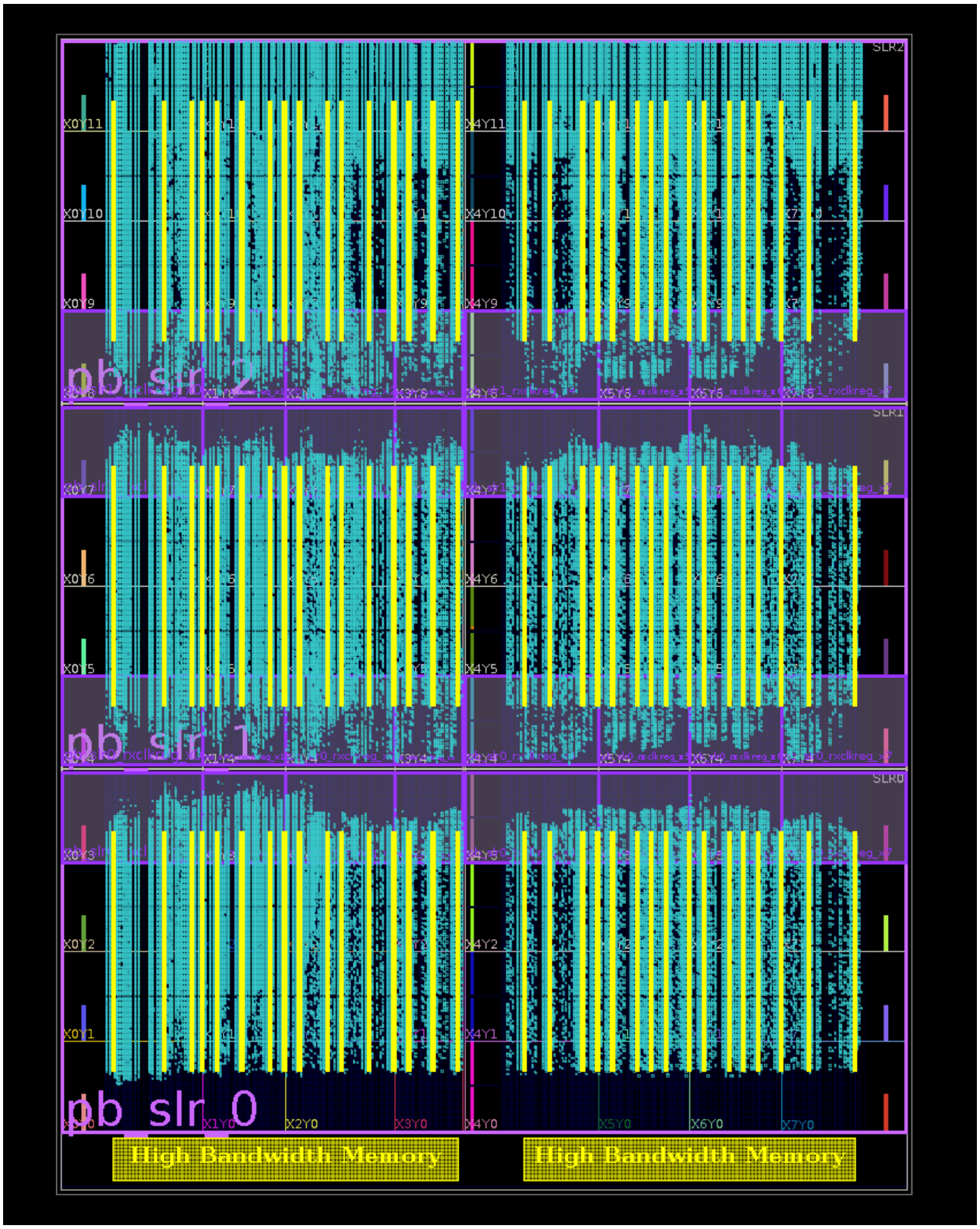
The package includes Tcl scripts that are used in the Vivado® software for precise control of resource placement for DSP SuperTile arrays, control of resource placement and routing at SLR crossings, and timing constraint control by specifying multi-cycle paths. For a technical description of the DSP Supertile, refer to *A High-Throughput Reconfigurable Processing Array for Neural Network* [Ref 1].

Implementation Results on XCVU37P-2E

Physical Placement Picture and Description

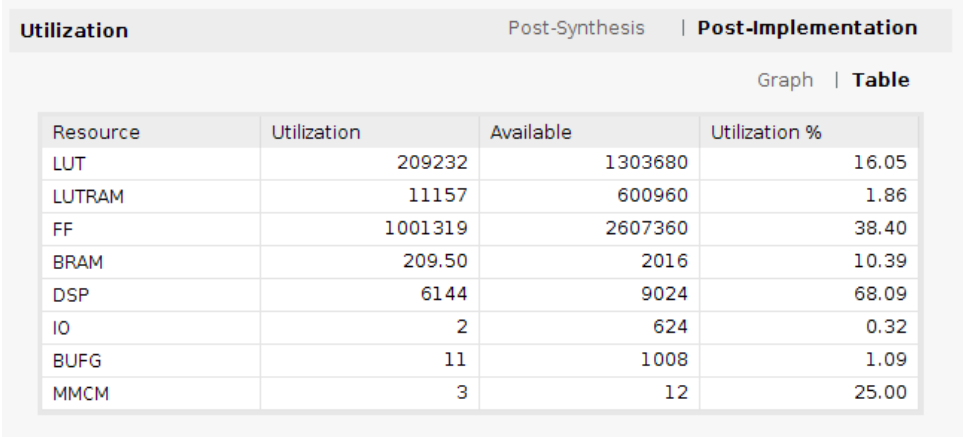
An example implementation of a *32×192* multiplier array that spans all 3 SLRs of the XCVU37P-2E FPGA is shown in the following figure. This example uses 64 DSP rows in each SLR.

Figure 2: **View of a Vivado Implementation of a 32x192 DSP Array on XCVU37P-2E FPGA**



The maximum operation frequency of this design on the XCVU37P-2E FPGA is 638 MHz. The implementation resource usage data of this design is shown in the following figure.

Figure 3: **XCVU37P-2E FPGA Resource Utilization**



Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	209232	1303680	16.05
LUTRAM	11157	600960	1.86
FF	1001319	2607360	38.40
BRAM	209.50	2016	10.39
DSP	6144	9024	68.09
IO	2	624	0.32
BUFG	11	1008	1.09
MMCM	3	12	25.00

Reference Example Data (Inputs and Expected Output) in SRAM

To demonstrate correctness, example cases for *M×V* command and data inputs as well as output reference data are listed in the following table.

Table 2: **Reference Example Test Cases Provided in SRAM Intialization**

Test Case	*A* type	*B* type	*M*	*K*	*N*
1	int8	int8	128	384	32
2	uint8	int8	128	384	32
3	int8	uint8	128	384	32
4	uint8	uint8	128	384	32

Output Signals: Done and Pass/Fail

As part of the reference implementation, the *M×V* block provides an output pass/fail indicator signal after making a comparison of the generated output data with the provided reference output values.

Reference Design

Download the reference design files for this application note from the Xilinx® website.

Reference Design Matrix

The following checklist indicates the procedures used for the provided reference design.

Table 3: **Reference Design Matrix**

Parameter	Description
General	
Developer name	Xilinx
Target devices	XCVU37P-2E FPGA

Table 3: **Reference Design Matrix** *(cont'd)*

Parameter	Description
Source code provided?	Yes
Source code format	SystemVerilog
Design uses code or IP from existing reference design, application note, third party or Vivado® software?	ChipScope™ VIO
Simulation	
Functional simulation performed	Yes
Timing simulation performed?	No
Test bench provided for functional and timing simulation?	Yes (Functional simulation)
Test bench format	SystemVerilog
Simulator software and version	XSIM 2019.1
SPICE/IBIS simulations	No
Implementation	
Synthesis software tools/versions used	Vivado® synthesis
Implementation software tool(s) and version	Vivado Implementation 2019.1
Static timing analysis performed?	Yes

Conclusion

This application note presented a reference design for a multi-precision integer matrix-vector multiplier (MxV). The reference design on XCVU37P-2E FPGA supports both 8-bit and 16-bit input operands, as well as 32-bit and 64-bit accumulators. Although the XCVU37P-2E FPGA consists of three SLRs, the MxV operates as if the FPGA were a monolithic device running at 638 MHz. Such a design achieves the highest throughput and the lowest latency for a given amount of DSP resources on multi-SLR devices. In 8-bit mode, the MxV achieves a throughput of 15.7 Top/sec, and in 16-bit mode, the throughput is 7.8 Top/sec. The reference design includes not only the HDL design and Vivado® tools physical design scripts, but also Python scripts for verifying block-matrix multiplication. This reference design can therefore become an integral part of a low-latency, high-throughput reconfigurable neural-network accelerator.

References

These documents provide supplemental material useful with this document:

1. E. Wu, X. Zhang, D. Berman and I. Cho, *A high-throughput reconfigurable processing array for neural networks*, in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Ghent, 2017.
2. *The Anatomy of an Embedded Machine Learning Accelerator* (WP515)
3. *FPGAs in the Emerging DNN Inference Landscape* (WP514)
4. E. Wu, X. Zhang, D. Berman, I. Cho and J. Thendean, *Compute-Efficient Neural-Network Acceleration*, in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*, New York, 2019.

5. *Deep Learning with INT8 Optimization on Xilinx Devices* (*Deep Learning with INT8 Optimization on Xilinx Devices* (WP486)
6. *Embedded Vision with INT8 Optimization on Xilinx Devices* (WP490)
7. S. V .D. Walt, S. C. Colbert and G. Varoquaux,*The NumPy Array: A Structure for Efficient Numerical Computation*, *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22-30, 2011.

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
02/27/2020 Version 1.0	
Initial release.	N/A

Please Read: Important Legal Notices

Copyright