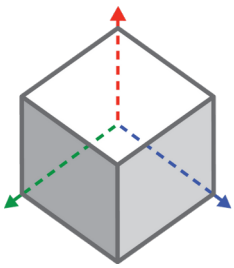

VectorBlox
embedded supercomputing

VectorBlox MXP Programming Guide for Intel

August 07, 2018

VectorBlox

http://vectorblox.com

Contents

1 VectorBlox MXP

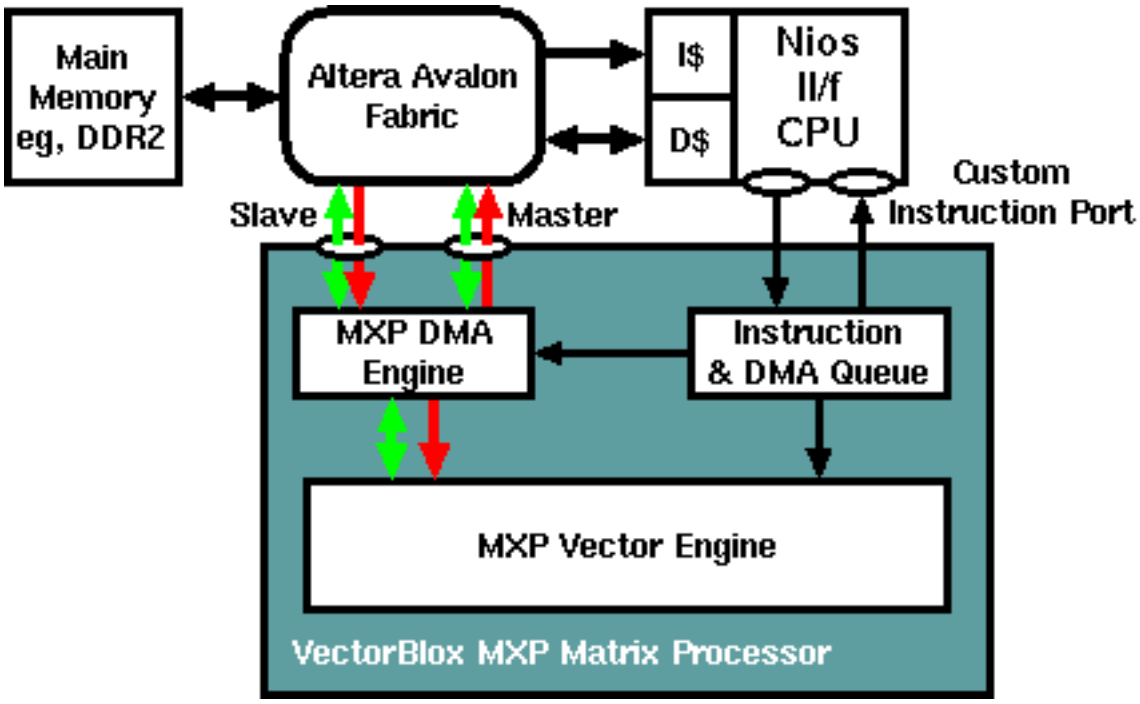

Figure 1: VectorBlox MXP System with Nios II

1.1 Architecture Overview

The VectorBlox MXP matrix processor is an extremely high-performance processor capable of speedups in excess of 1000× faster than MicroBlaze or Nios II/f. The design of the processor was inspired by the vector processors used in scientific supercomputers made by Cray, Fujitsu and NEC. However, the MXP is not a simple clone of one of these processors. It has been redesigned from the ground up to perform well on embedded applications which operate primarily on various integer widths and fixed-point data types. It has also been designed from the start to map exceptionally well into modern FPGAs in a way that exploits their hard RAM blocks and hard multiplier or DSP blocks.

The VectorBlox MXP matrix processor is an update on the classical vector processor. Instead of operating merely on vectors, it can also operate on 2D and 3D matrices. It performs parallel calculations directly on sets of data stored directly in a private scratchpad memory, rather than a register file.

To achieve speedups in excess of 1000×, the VectorBlox MXP employs several strategies to maximize parallelism and to reduce overhead such as address calculations. This ensures the parallel ALUs employed by the MXP are working to their full potential on actual data calculations.

The parallelism available in MXP exceeds that of traditional scalar CPUs, fixed-width SIMD operations, and even variable-length vector CPUs. This goes beyond the number of parallel ALUs employed, as we have witnessed speedups of 20× or higher using just a single vector lane! Here are some of the reasons why you may expect speedups that exceed the number of ALUs on your own code:

- Unlike scalar CPUs or fixed-width SIMD operations, loop counters and branches are not necessary and can be eliminated from the inner loop. This eliminates the overhead of counting/incrementing, comparisons, conditional branches, and branch mispredictions.

- Unlike traditional scalar vector CPUs, each ALU in MXP can perform 2 parallel calculations on halfwords (16-bit integers) or 4 parallel calculations on bytes (8-bit integers).
- Unlike scalar CPUs or fixed-width SIMD operations, load and store instructions are not necessary and can be eliminated from the inner loop. These normally transfer data from a cache to the register file, and may even result in a cache miss. Instead, the MXP operates memory-to-memory on data already in the scratchpad and never suffers from a cache miss.
- Traditional vector CPUs support a fixed number of named vectors, each with a maximum size. Instead, the MXP can partition its large and flexible scratchpad arbitrarily into any number of vectors, of any length, starting at any address, that is subject only to overall scratchpad capacity. This improves overall data availability and significantly reduces data copying.
- The MXP scratchpad is easily double-buffered, allowing all memory latency to be hidden.

1.2 Scratchpad

The MXP does not operate on data directly stored in memory. Instead, data must first be DMA-transferred into a private local memory called a scratchpad. Like a cache, the scratchpad is designed to provide fast, parallel access to data. However, unlike a cache, the scratchpad is not managed automatically for the programmer. Instead, the programmer must explicitly transfer data from host to scratchpad, or from scratchpad to host.

The scratchpad is byte addressable. A vector, matrix, or submatrix is identified explicitly by a pointer to its starting address in the scratchpad. The contents of a vector are striped across several parallel block memories, allowing full parallel readout when a vector is accessed sequentially. All vector operations start at the lowest address and proceed to the highest address, subject to the length of the vector. Operations on long vectors are carried out over multiple clock cycles, one ***wavefront*** of data at a time. No matter what the starting address (aligned or not), a full wavefront which spans the full width of the scratchpad can always be read in one clock cycle.

1.3 Data Organization: Vectors, 2D Matrices and 3D Matrices

The most efficient way to use MXP is to organize data in memory contiguously into vectors, or as packed 2D or packed 3D matrices.

By a 2D packed matrix, we mean a series of rows of data that are placed end-to-end, possibly with some fixed amount of padding between each pair of adjacent rows. There must be a constant difference between the starting addresses of any two adjacent rows.

By a 3D packed matrix, we mean a series of 2D matrices that are placed end-to-end, possibly with some fixed amount of padding between each pair of 2D adjacent matrices. There must be a constant difference between the starting addresses of any two adjacent 2D matrices.

Before operating on any data, the MXP processor must also be told the size or dimensions of the vector or matrix. This information is provided to the processor in advance of the instructions and remembered in its configuration state.

The minimal information required for vector instructions is the ***vector length***. As will be described later, more information is also required for 2D and 3D matrix operations, such as the number of rows.

1.4 Vector Instructions

A typical instruction is provided with three explicit operands: pointers to the destination, to the source operand A, and to the source operand B. In addition, type information is explicitly provided to specify the data element size (byte, halfword or word), signed or unsigned operation, as well as special cases for operands A and B. The special

cases that are permitted are replacing vector operand A with a scalar value, and replacing vector operand B with an enumerated vector. An enumerated vector provides an ordinal number for each element indicating its position in the vector.

1.5 Vector Lanes

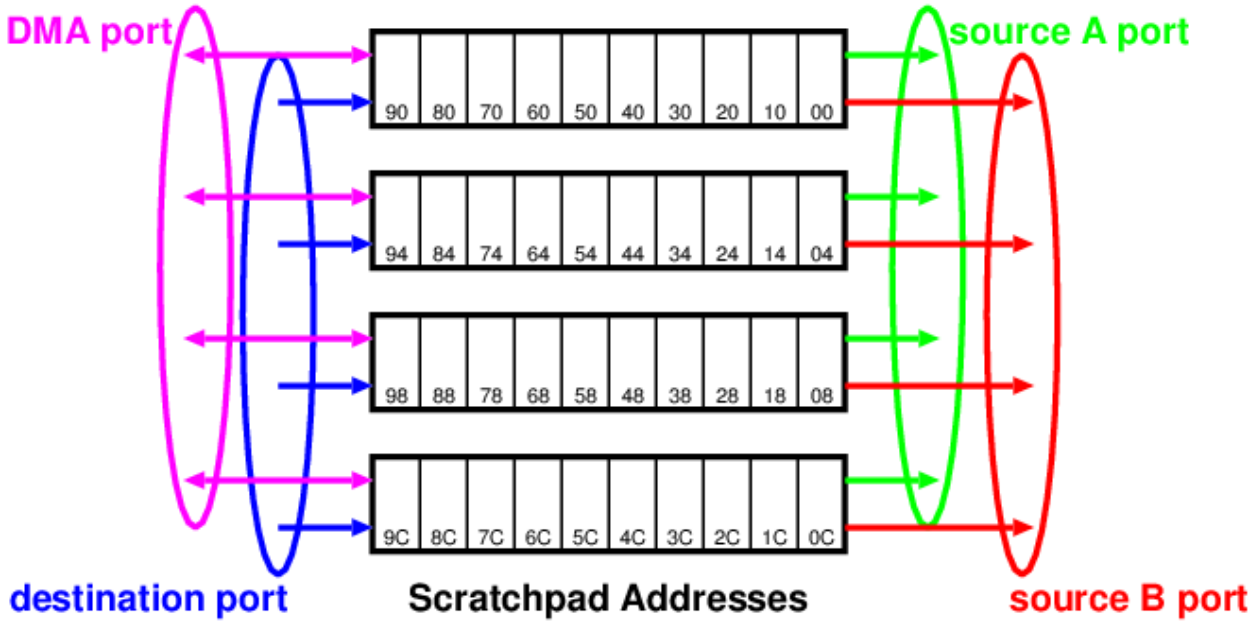

Figure 2: Quadruple-ported Scratchpad and Addressing

Calculations within MXP are performed by parallel 32-bit vector lanes. A processor configuration that is referred to as MXP-V4, for example, consists of four parallel vector lanes. Likewise, a V1 contains just one vector lane and V16 contains sixteen vector lanes.

The vector lanes can be instructed to operate upon three different data sizes: bytes (8 bits), halfwords (16 bits), or words (32 bits). When operating on smaller data sizes, the amount of parallelism increases proportionately. That is, a V4 configuration can perform either 16 parallel byte-size operations per cycle, or 8 parallel halfword-size operations per cycle.

Each lane contains a slice of the scratchpad memory, so wider vector processors are provided with a wider scratchpad memory. This scales the available memory bandwidth in a natural way to match the compute capacity of the MXP vector lanes.

As shown in the figure above, the scratchpad contains four access ports, and each is byte-addressable. Two dedicated read ports and one dedicated write port allows the processor to read its operands and write back a result every clock cycle. The fourth port, a DMA port, can be dynamically configured to either read or write data. The DMA system can access the scratchpad without interrupting a computation in progress.

Furthermore, addresses in the scratchpad are striped across the vector lanes, in a similar way that data blocks are striped across disks in RAID-0. Each lane can store one word, two halfwords, or four bytes. Hence, scratchpad addresses increase by 4 when crossing from one lane to an adjacent lane.

Unlike a typical register-based processor, there is no preset limit imposed by the MXP instruction set architecture (ISA) on the number of vectors or the vector length. However, the size of the scratchpad forms a practical upper bound on the vector length and various other matrix parameters. Hence, at one extreme, the entire scratchpad can be filled by a single vector of bytes. At the other extreme, the entire scratchpad can be filled entirely by vectors that are each one byte long.

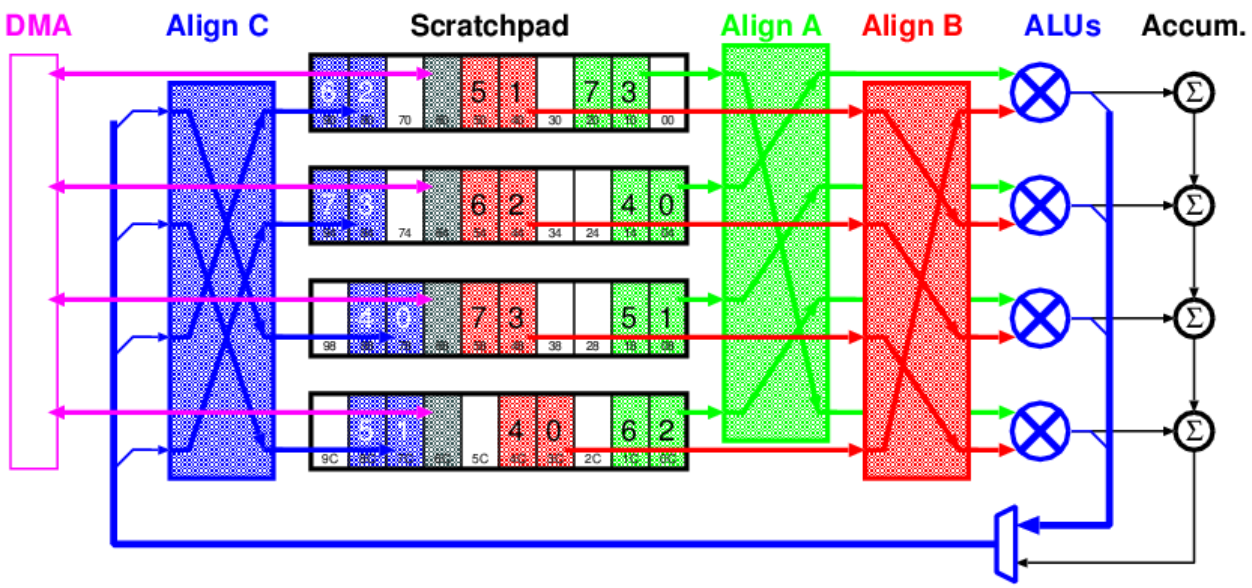

Figure 3: VectorBlox MXP Vector Engine

While the ISA does not limit the number of vectors stored in the scratchpad, it is important to note that all scratchpad pointers are calculated and stored by the host scalar processor. Since the scalar register file is fixed in size and the operation of many registers are predetermined by its compiler application binary interface (ABI) specifications, using more than 8 to 16 vectors will likely involve spilling contents of the scalar register file to main memory.

2 Programming Model

Programs written for the VectorBlox MXP use a combination of ANSI-C with VectorBlox C extensions. Basic ANSI-standard C code is run entirely on the host scalar processor. The VectorBlox C extensions are used to to specify the data-parallel operations that are to be computed by the MXP vector engine. These extensions are especially useful for image processing and similar applications where operations are applied to sets of data (such as pixels). The VectorBlox MXP Matrix Processor provides two engines that run completely in parallel with the host processor: the MXP DMA engine, and the MXP vector engine.

2.1 MXP DMA Engine

The MXP direct-memory access (DMA) engine provides a means to perform block data transfers between the host's external memory and the vector scratchpad. These transfers are performed asynchronously from the host processor.

The contents of the scratchpad are not managed automatically like a cache. Instead, the programmer must explicitly transfer memory from the host to the vector scratchpad, or vice versa.

Only one DMA transfer can be active at a time. Additional requests are queued by the system until the active transfer is complete. To ensure correctness, DMA transfers are always executed in FIFO or program order. Since DMA transfers can be interleaved with vector instructions, they are deposited into a common ***Instruction and DMA Request Queue***. The MXP processor will automatically allow DMA transfers to bypass in-flight instructions, or vice versa, provided there are no data hazards between them. These data hazards are determined exclusively by the scratchpad address(es) being read or written. When a hazard exists, the MXP processor will insert a bubble into the pipeline, allowing current operations to complete before allowing a hazardous operation to make forward progress.

2.2 MXP Vector Engine

The MXP vector engine operates naturally on 8 bit bytes, 16 bit halfwords, or 32 bit words. The engine is organized into a set of vector lanes, where refer to the size of the vector engine as V1 or V128 for 1 or 128 vector lanes, respectively. The number of lanes must be a power of 2, where each lane incorporates a 32-bit ALU and a 32-bit slice of the scratchpad. Each vector lane can be subdivided on an instruction-by-instruction basis into two halflanes which operate on halfwords, or four bytelanes which operate on bytes. Thus, maximum parallelism is provided on byte-wide data.

The vector lanes of the MXP vector engine are only one means of achieving parallel execution. Even a single-lane MXP V1 with one 32-bit ALU can provide speedups over 20x compared to the host processor. This parallelism may come from many sources:

- byte or halfword parallelism
- elimination of load and store instructions from the instruction stream
- double-buffered asynchronous DMA operations overlap memory latency with computation
- hardware-based loop counters and auto-incrementing address arithmetic
- hardware-based looping avoids branch mispredictions
- overlapped scalar instructions with vector instructions and DMA operations

2.3 MXP Programming Overview

The general process for programming the MXP is:

1. Allocate vectors in scratchpad.
2. Transfer data from memory to scratchpad.
3. Operate on vectors in scratchpad.
4. Transfer data from scratchpad to memory.
5. Deallocate vectors in scratchpad.

The scratchpad is a region of fast, on-chip parallel memory for holding and operating upon vector data. A minimum of 4KB per vector lane is provided, and this is usually enough for most applications.

The scratchpad is addressable by the scalar host, but care must be taken whether to cache the scratchpad or not. By default, scratchpad addresses are cached.

A simple vector program which adds three vectors is provided below:

2.3.1 Simple Example

```
#include "vbx.h"

int A[] = {1, 2, 3, 4};
int B[] = {5, 6, 7, 8};
int C[] = {-1, -1, -1, -1};
int main()
{
    int vector_len = 4;
    int num_bytes = vector_len * sizeof(int);

    /* step 1 */
    vbx_word_t * v_a = vbx_sp_malloc( num_bytes );
    vbx_word_t * v_b = vbx_sp_malloc( num_bytes );
    vbx_word_t * v_c = vbx_sp_malloc( num_bytes );

    /* step 2 */
    vbx_dma_to_vector( v_a, A, num_bytes );
    vbx_dma_to_vector( v_b, B, num_bytes );

    /* step 3 */
    vbx_set_vl( vector_len );
    vbx( VVW, VADD, v_c, v_a, v_b );

    /* step 4 */
    vbx_dma_to_host( C, v_c, num_bytes );

    /* step 5 */
    vbx_sp_free();

    vbx_sync();
    printf( "C[] = %d, %d, %d, %d\n",
    C[0], C[1], C[2], C[3] );

    return 0;
```

```
}
```

This example could possibly run into some caching issues, information on how to avoid these issues, as well as using dynamic allocation can be found in the data-sharing section.

3 Nios II Programming

The Nios II/f host processor has the following properties:

- 6-stage pipeline, roughly 1 instruction per clock cycle
- single-issue, in-order execution
- dynamic branch prediction
- direct-mapped, non-coherent instruction and data caches
- 2GB address space (without MMU), bit-31 cache bypass
- hardware support for multiply (1 cycle operation)
- hardware support for divide (from 4 to 66 cycles per operation)
- barrel shifter
- optional Memory Management Unit (MMU) or Memory Protection Unit (MPU)

VectorBlox typically configures the Nios II/f in its demonstration systems as follows:

- 8KB instruction cache, burst transfers enabled
- 4KB data cache, 32-byte line size, burst transfers enabled
- no MMU or MPU

3.1 Nios II Caching

The Nios II does not provide any mechanism for hardware cache coherence. Hence, programmers must manually flush the instruction or data cache when necessary. Flushing consists of writing back a data cache line (if dirty), and then invalidating the cache line.

Altera provides two ways to flush the data cache:

- `#include <sys/cache.h>` required header file.
- `void alt_dcache_flush_all(void)` flushes the entire data cache.
- `void alt_dcache_flush(void *start, alt_u32 len)` flushes the specified memory region from the data cache.

You can replace `dcache` with `icache` to flush the instruction cache.

Note that `alt_dcache_flush()` must walk through the entire memory region of `len` bytes (incrementing by the cache line size in the inner loop). For a sufficiently large memory region, it will be faster to just flush the entire cache and start over.

> For example, a typical 8KB data cache with 32b line size contains 256 cache lines. Invalidating an 8MB memory region will require flushing each cache line 1024 times. Since it takes roughly 20 clock cycles to refill a flushed cache line, it is better to flush the entire cache and start over.

On a Nios II/f processor, setting address bit 31 will bypass the cache.

> Nios II/s and II/e have no data cache and force this bit to 0.

The functions below convert a pointer between cached and uncached mode by toggling this bit:

- `#include <sys/cache.h>` required header file.
- `void *alt_remap_cached (volatile void *ptr, alt_u32 len)` returns a cached pointer.
- `volatile void *alt_remap_uncached (void *ptr, alt_u32 len)` returns an uncached pointer. When mapping to uncached, the memory region will also be flushed.

To allocate/deallocate uncached memory:

- `#include <sys/cache.h>` required header file.
- `volatile void *alt_uncached_malloc (size_t size)` allocates memory, returns uncached pointer.
- `void alt_uncached_free (volatile void *ptr)` unallocates an uncached region.

3.2 VBX Portability Library

Rather than write Nios-specific code, VectorBlox provides a cache-management API that closely matches the Nios API. This helps a VectorBlox MXP program to be more easily ported to other scalar CPU hosts.

The functions in the portability library are given in table. More details can be found in the *VectorBlox MXP Programming Reference*.

- `vbx_dcache_flush(PTR,len)` considers the length of data being flushed. If it is too large, it will flush the entire data cache instead. Altera's version loops over the entire length of the data.
- `vbx_remap_uncached(PTR)` remaps and flushes only a single cache line
- `vbx_remap_uncached_flush(PTR,len)` remaps and flushes a region, but the region is flushed using `vbx_dcache_flush(PTR,len)`

Altera Library	VBX Equivalent
`alt_timestamp_type`	`vbx_timestamp_t`
`alt_timestamp_start()`	`vbx_timestamp_start()`
`alt_timestamp_freq()`	`vbx_timestamp_freq()`
`alt_timestamp()`	`vbx_timestamp()`
`alt_uncached_malloc()`	`vbx_uncached_malloc()`
`alt_uncached_free()`	`vbx_uncached_free()`
`alt_dcache_flush_all()`	`vbx_dcache_flush_all()`
`alt_dcache_flush(PTR,len)`	`vbx_dcache_flush(PTR,len)` or `vbx_dcache_flush_line(PTR)`
`alt_remap_cached(PTR,len)`	`vbx_remap_cached(PTR,len)`
`alt_remap_uncached(PTR,len)`	`vbx_remap_uncached(PTR)` or `vbx_remap_uncached_flush(PTR)`

4 Data Sharing

The code in the following subsections are variations of the vector addition code given earlier (simple example) using the VBX Portability Library.

The original code has no caching issues because the data is statically allocated in the data segment, so it is initialized into the data segment and untouched by the processor. However, the new code uses the processor to dynamically allocate and initialize the data, hence a copy of the data may be residing in the cache.

However, the first example shows one approach of managing cache coherence by flushing the shared regions out of the data cache. The source memory regions, `A` and `B`, must be flushed before the DMA transfer because the CPU has modified their values and dirty lines are being retained in the write-back cache. The memory region for the final answer, `C`, must be flushed because a stale copy or dirty copy may be in the data cache. Even if the CPU did not initialize the array `C`, elements at the beginning or end of C may have been brought into the cache because of locality and the sharing of cache lines with adjacent data. As a result, it can be difficult to keep track of whether to flush the data.

Instead, the second example shows our preferred approach of managing cache coherence. It works by marking shared regions as uncached data during allocation. Then, all scalar accesses will be to uncached data, but there will not be any coherence issues. This is the easiest way to program, but it may result in performance issues when the scalar processor initializes shared inputs or reads shared results. We still recommend this style, but suggest that you address performance issues by explicitly switching to a cached pointer where necessary. This is shown for array `A` in the final example.

4.1 Data Sharing Examples

4.1.1 Example showing flushing of cached regions

```
#include "vbx.h"
#include "vbx_port.h"

int main()
{
    int A[] = {1, 2, 3, 4};
    int B[] = {5, 6, 7, 8};
    int C[] = {-1, -1, -1, -1};

    int vector_len = 4;
    int num_bytes = vector_len * sizeof(int);

    /* step 1 */
    vbx_word_t *va = vbx_sp_malloc( num_bytes );
    vbx_word_t *vb = vbx_sp_malloc( num_bytes );
    vbx_word_t *vc = vbx_sp_malloc( num_bytes );

    /* step 2 */
    vbx_dcache_flush( A, num_bytes );
    vbx_dcache_flush( B, num_bytes );
    vbx_dma_to_vector( va, A, num_bytes );
    vbx_dma_to_vector( vb, B, num_bytes );

    /* step 3 */
    vbx_set_vl( vector_len );
    vbx( VVW, VADD, vc, va, vb );
```

```
    /* step 4 */
    vbx_dcache_flush( C, num_bytes );
    vbx_dma_to_host( C, vc, num_bytes );

    /* step 5 */
    vbx_sp_free();

    printf( "C[] = %d, %d, %d, %d\n",
            C[0], C[1], C[2], C[3] );
    return 0;
}
```

4.1.2 Example with shared regions allocated as uncached

```
#include "vbx.h"

int main()
{
    int vector_len = 4;
    int num_bytes = vector_len * sizeof(int);

    int *A; A = vbx_shared_malloc( num_bytes );
    int *B; B = vbx_shared_malloc( num_bytes );
    int *C; C = vbx_shared_malloc( num_bytes );

    A[0] = 1; A[1] = 2; A[2] = 3; A[3] = 4;
    B[0] = 5; B[1] = 6; B[2] = 7; B[3] = 8;

    /* step 1 */
    vbx_word_t *va = vbx_sp_malloc( num_bytes );
    vbx_word_t *vb = vbx_sp_malloc( num_bytes );
    vbx_word_t *vc = vbx_sp_malloc( num_bytes );

    /* step 2 */
    vbx_dma_to_vector( va, A, num_bytes );
    vbx_dma_to_vector( vb, B, num_bytes );

    /* step 3 */
    vbx_set_vl( vector_len );
    vbx( VVW, VADD, vc, va, vb );

    /* step 4 */
    vbx_dma_to_host( C, vc, num_bytes );

    /* step 5 */
    vbx_sp_free();

    printf( "C[] = %d, %d, %d, %d\n",
            C[0], C[1], C[2], C[3] );
    return 0;
}
```

4.1.3 Example combining shared (uncached) and cached regions

```c
#include "vbx.h"
#include "vbx_port.h"

int main()
{
    int vector_len = 4;
    int num_bytes = vector_len * sizeof(int);

    int *A; A = vbx_shared_malloc( num_bytes );
    int *B; B = vbx_shared_malloc( num_bytes );
    int *C; C = vbx_shared_malloc( num_bytes );

    int *cachedA = vbx_remap_cached( A, num_bytes );
    cachedA[0] = 1; cachedA[1] = 2;
    cachedA[2] = 3; cachedA[3] = 4; // cached, faster
    vbx_dcache_flush( cachedA, num_bytes );
    B[0] = 5; B[1] = 6;
    B[2] = 7; B[3] = 8; // uncached, slower

    vbx_word_t *va = vbx_sp_malloc( num_bytes );
    vbx_word_t *vb = vbx_sp_malloc( num_bytes );
    vbx_word_t *vc = vbx_sp_malloc( num_bytes );

    vbx_dma_to_vector( va, A, num_bytes );
    vbx_dma_to_vector( vb, B, num_bytes );

    vbx_set_vl( vector_len );
    vbx( VVW, VADD, vc, va, vb );

    vbx_dma_to_host( C, vc, num_bytes );

    vbx_sp_free();

    printf( "C[] = %d, %d, %d, %d\n",
             C[0], C[1], C[2], C[3] );
    return 0;
}
```