VCOP Kernel-C to C7000 Migration Tool

User's Guide

Literature Number: SPRUIG3C
January 2018–Revised August 2019

Contents

Read This First

About This Manual

This document describes a tool which accepts kernels written for EVE's VCOP processor in the VCOP Kernel-C language, and rewrites them as a C program suitable for execution on C7000™ (C7x). This tool is intended to assist programmers who have EVE code in migrating to C7000.

Related Documentation

The following documents will provide related information for the C7x:

- *C7000 C/C++ Optimizing Compiler User's Guide* (SPRUIG8)
- *C7000 Host Emulation User's Guide* (SPRUIG6)
- *C7000 Embedded Application Binary Interface (EABI) Reference Guide* (SPRUIG4)
- *C6000-to-C7000 Migration User's Guide* (SPRUIG5)

Trademarks

C7000 is a trademark of Texas Instruments.
OpenCL is a trademark of Apple Inc. used with permission by Khronos.

Overview and Scope

This document describes tooling that enables the migration of kernels written for EVE's VCOP vector coprocessor to the C7000 (C7x).

Topic **Page**

1.1 Comparing VCOP and C7000

VCOP kernels are written in a domain-specific language subset of C++ called VCOP Kernel-C. A migration tool accepts kernels written in Kernel-C and outputs code that will be functionally equivalent, within limits, for C7x. This document provides details about the VCOP Kernel-C migration tool: its scope, capability, and limitations.

VCOP Kernel-C, compiled for native execution on EVE, is translated to EVE code by a tool called **vcc-arp32**. (*VCC* stands for VCOP C Compiler.) The vcc-arp32 tool generates kernel code for VCOP, as well as C code for ARP32 that sets up the execution environment for the kernel such that the kernel can be called as a C function. This code is then compiled by the ARP32 compiler for native execution on EVE.

For C7x the VCOP Kernel-C migration tool is called **vcc7x**. It translates a kernel written in VCOP Kernel-C into a C function that can be compiled for C7x and called in the same way as on ARP32.

Typically the kernel is called by dispatch code running on the ARP32 that manages buffers, transfers data, and synchronizes execution between VCOP and ARP32. The APIs that provide this functionality, which are part of the ARP32 RTS, are re-implemented for C7x so that dispatch code can be migrated with little or no modification.

The migration tool is supported by a library that implements specific VCOP operations on C7x. The library is called the VCOP Virtual Machine (*VVM*, or simply *virtual machine*) since it layers a VCOP-like interface onto the underlying C7x. VCC translates Kernel-C statements into calls to the virtual machine's APIs. The VVM is implemented as a C++ library that uses template classes and inline functions to achieve abstraction and efficiency.

On the surface VCOP and C7x have many similarities in that both have SIMD capability in the form of vector-oriented datapaths. In addition, both are designed for iterative (rather than thread-level) operation using pipelined execution.

However, there are also many differences.

- VCOP has 8-way SIMD with 40-bit integer arithmetic in each lane for a total of 320 bits of arithmetic width, while C7x has 512 bits of width with variable lane widths arranged as 64 lanes x 8 bits, 32x16, 16x32, or 8x64.
- VCOP has built-in control for up to four levels of nested looping, whereas C7x relies on traditional explicit loop control and software pipelining.
- VCOP has built-in address generators that automatically compute multi-dimensional array accesses; C7x has the Streaming Engine (SE) and Streaming Address Generators (SA), which are similar concepts but differ in the details.
- VCOP has built-in support for lookup-table and histogram operations; C7x also supports these operations but not as directly.
- VCOP has a rich set of addressing modes that enable loads and stores to perform upsampling, downsampling, interleaving, scatter/gather, and so on. C7x has support for many of these operations but others require software emulation.
- VCOP has 96K bytes of L1 memory for buffers, lookup tables, and histogram bins that must be manually loaded and unloaded; C7x has 32K bytes of L1D SRAM (in addition to an L1 data cache that is automatically managed).
- Finally, VCOP has a parameter buffer that allows kernel-invariant computations to be pre-computed prior to execution of the kernel, whereas C7x has no such dedicated support, although such a mechanism can be implemented programmatically.

All these differences must be accounted for when translating code written for VCOP to C7x.

1.2 About this Document

The purpose of this document is to describe the behavior of the VCOP Kernel-C migration tool and the supporting virtual machine. Specifically, it describes how each feature of VCOP is implemented on C7x by the migration tool. After reading this document, a user should be able to look at either the C code generated by the migration tool or the assembly output that results from compiling it and understand what it's doing and why.

Compatibility Warnings:
The primary goal of the translation capability is functional correctness. That is, a kernel written for VCOP should execute unmodified on C7x with identical (bit-exact) results as on VCOP. However, there are cases where this is not possible or impractical. In these cases, the kernel or calling code may need to be modified to execute correctly. These cases are highlighted as Compatibility Warnings in this document, and summarized in Section A.1.

Efficiency Warnings:
Most VCOP constructs map well to C7x with little or no loss of efficiency. However there are a few cases where efficient translation is not possible. These cases are highlighted as Efficiency Warnings and summarized in Section A.2.

1.2.1 Documentation Conventions

The following terms are used throughout the document:

- *VCC* refers generically to either the VCOP Kernel-C compiler (vcc-arp32), or the VCOP Kernel-C migration tool (vcc7x).
- *Migration tool* refers to vcc7x, and sometimes implicitly includes the virtual machine.
- Function names `init()`, `kernel()`, and `vloops()` refer to the functions generated by the VCC tool to implement the kernel API. The real names are prepended with the name of the kernel. For example, for a kernel named `image_filter` the term "`vloops()` function" in this document refers to the `image_filter_vloops()` function generated by VCC.
- The term *tvals structure* refers to the VCC-generated structure that holds values computed in the `init()` function and used by the `vloops()` function, corresponding to VCOPs pblock. See Section 2.2.1. An expression of the form `tvals->pN` refers to an arbitrary field from the tvals structure.
- Symbols in angle brackets are placeholders for one of several values of the indicated type; for example <*op*> represents some operation, <*type*> represents some type, and so on.

In code sequences in either the text or the tables, the following symbolic conventions are used:

- A symbol starting with an upper-case V is either a VCOP or C7x vector register or a variable that represents such a register: `Vsrc, Vdst, Vtmp, Vpred`, etc.
- An operand starting with `Vperm` is a C7x vector register containing a pre-computed vector for use with a VPERM instruction.
- An operand starting with upper-case S is a C7x scalar variable or register: `Sreg, Stmp`, etc.
- An operand starting with upper-case C is a C7x scalar constant: `Cwidth, Csize`, etc.
- An operand starting with upper-case P is a C7x vector predicate: `Pred, Pnz`, etc. Pmask is a special vector predicate used to select certain lanes. Symbols of the form `P8b, P8h, P8w` and so on are precomputed vector predicates used as lane masks for partial vector stores; for example `P8h` is a mask for storing 8 halfwords. See Section 5.2.
- `Addr` represents a C7x indirect address expression, perhaps with an register or SA-based offset or index, for example `*A4`, `*A4[A0]`, or `*B0[SA0++]`.
- Symbols `base, in`, and `out` represent the base address in a VCOP load or store. A base address can be an expression, or a value loaded from the tvals structure (see Section 2.2).
- A symbol that begins with `Agen` represents either a VCOP address generator declared with `__agen`, or the C7x variable or register that represents its translation.

  `SEn` refers to a C7x Streaming Engine access, either SE0 or SE1. Similarly `SAn` refers to a C7x Streaming Address Generator access.

1.3 Output Format

On EVE, the vcc-arp32 tool translates a kernel to a C source file that contains the five functions described in Chapter 2, which is then compiled by the ARP32 compiler. The `vloops()` function contains VCOP intrinsics to implement the vector loop command(s).

On C7x, the vcc7x tool translates a kernel to C++ source file that contains the same five functions, which is then compiled by the C7x compiler.

It is feasible to use this translated output as the basis for additional development or performance optimization by directly modifying the generated code. Under this scenario, translation is a one-time step and the generated code becomes the source code. However, it's important to be aware of the limitations described in this section. Whether a user chooses to modify the translated output as the basis for further development is up to the user.

Both versions of VCC invoke the C preprocessor on the Kernel-C input, which removes comments, handles preprocessor directives like `#if` and `#include`, and expands macros. The implication is that the C/C++ output of the migration tool will have these elements removed, with the exception of `VCOP_SIMD_WIDTH`. (Additionally, `VCOP_SIMD_WIDTH` will not be expanded in #if statements.) The translated output will therefore be less readable and less configurable than the source.

The migration tool makes some effort to make the translated output resemble the original Kernel-C source code. In particular:

- Variable names (vectors, agens, and loop counters) are preserved.
- Original indentation and white spacing is not preserved, but the migration tool applies its own output formatting to make the output readable.
- VCOP statements that map to simple C expressions are generated that way. Other statements generally map to function calls in the VCOP virtual machine so that the kernel function remains uncluttered.

VCOP vectors are translated using "native vector types" on C7x, which are built-in types in the C7x C Compiler that are declared using OpenCL™ syntax. The typedef `__vector` maps to an OpenCL type corresponding to a VCOP vector.

The virtual machine is implemented in C++ using template functions and classes, allowing for a relatively economical yet efficient implementation. Since the virtual machine API is a C++ API, the generated kernel is a C++ source file and must be compiled as C++.

The virtual machine uses extensions to the C++ language, particularly native vector types and C7x intrinsics, and will therefore not be portable to other targets. In particular, the code will not compile and run on a PC or other GPP host.

Although the translated code is C++, the client code that calls it can be either C or C++. By default, the client code is assumed to be C and the migration tool includes `extern C` on the generate kernel functions (which are C++) so they are callable from C. If the `--cpp_out` option is used on the migration tool, the client code is assumed to be C++ and the `extern C` declarations are omitted.

1.4 Data Types

VCOP's vector registers have 8 lanes. Each stores 40 bits. Some arithmetic operations operate on all 40 bits of each lane. Others operate on fewer bits. Data in memory are signed or unsigned 8, 16, or 32 bit integers, which are sign or zero extended to 40 bits when loaded, and truncated or rounded when stored.

C7x has 512-bit vector registers, with variable-width lanes. That is, a vector register can hold 64 lanes of 8-bit data, 32 lanes of 32-bit data, and so on. To achieve efficient translation for the majority of cases, the migration tool will model VCOP's 40-bit arithmetic using 32-bit lanes. This can cause loss of precision for some kernels that depend on the extra 8 guard bits.

1.4.1 40-bit Incompatibilities

In general, there are two kinds of errors that result from narrowing lanes.

The first is when the upper bits indicate signedness. For example on VCOP the value 0x00.FFFF.FFFF represents a large unsigned number (4294967295), whereas the value 0xFF.FFFF.FFFF represents a negative number (-1). When translated, the 32 bit result 0xFFFF.FFFF could be either value depending on whether it's treated as signed or unsigned.

By default, the migration tool treats all 32-bit values as signed. This covers the majority of cases, since the 40-bit values in VCOP are always treated as signed. In some cases, however, this can lead to incorrect results. For example, when the value 0x00.FFFF.FFFF is right-shifted or compared, VCOP treats it as positive while the C7x translation treats it as negative.

To help address this issue, the migration tool will, in some cases, treat 32-bit values as unsigned. There are two ways this happens. First, if the vector is loaded from an unsigned base pointer (`__vptr_uchar`, `__vptr_ushort`, or `__vptr_uint`), its element type becomes unsigned. Second, the user can force a vector to become unsigned by declaring it using the special keyword `__vector_uint32` (rather than the normal `__vector`).

The following operations are affected by the signedness of vector elements.

- Right shifts (including shift-or) are unsigned if the left-hand-side operand is unsigned.
- Compare operations (including sort2, min, and max) use unsigned compares if both operands are unsigned; if only one operand is unsigned, these operations use signed compare.
- Saturation operations use unsigned compare for saturation bounds if the source register is unsigned.

Compatibility Warning: Unsigned vector elements
If a kernel relies on vector elements being treated as unsigned when bit 31 is set, the translated code may not work properly. Most such issues can be fixed by declaring the vector as __vector_uint32.

The second error that can result from the reduced lane width is when values have significance in the upper 8 bits. On VCOP these bits are typically used as overflow (guard) bits for accumulation loops, or to hold the upper bits of extended multiply operations. Here is a partial list of VCOP operations that use the guard bits:

- Multiplies (VMPY, VMADD, and VMSUB)
  Inputs are 17x17 (bit 16 is a sign bit) producing 40 bits of sign-extended output.
- VLMBD
  Searches bits 39-32 for leftmost bit.
- VADDH (Kernel-C: `(Vdst1,Vdst2) = Vsrc1 + hi(Vsrc2)`)
  Shifts guard bits right by 32 and adds them.
- VSHF16 (Kernel-C: `Vdst = jus16(src)`)
  Sign extends bit 32 into upper guard bits.

The migration tool does not attempt to account for or detect these incompatibilities. The resultant code will likely fail at run time.

Compatibility Warning: Reliance on 40-bit elements
If a kernel depends on more than 32-bits of precision in vector elements, the translated code may not work properly.

1.4.2 40-Bit Detection in Host Emulation Mode

To assist in detecting kernels that rely upon 40-bit elements, host emulation mode may be configured to warn when an operation is executed that depends on the high 8 bits. (For example, a comparison between two large values.) Defining `VCOP_40_BIT_DETECTION` to 1 will enable this feature. If `VCOP_40_BIT_FATAL` is also defined to 1, execution will halt after the first warning. This method of detection must be used in conjunction with a set of input data that is representative of expected input.

1.5 SIMD Width

VCOP has 8 lanes of 40 bits each. When mapped to 32-bit lanes on C7x, there are 16 lanes available, potentially doubling the throughput of a kernel.

Many kernels are written to be independent of the SIMD width, using the macro `VCOP_SIMD_WIDTH` to abstract the number of lanes. Some of these kernels can be successfully built in host emulation mode for wider (or narrower) machines simply by changing the value of the macro. In host emulation mode, `VCOP_SIMD_WIDTH` must now be defined on the command line or before inclusion of vcop_host_emulation.h.

The SIMD width used by VCC is controlled by the `--vcop_simd` option. (Kernels that qualify for SIMD 16 are NOT automatically detected or transformed.) For a SIMD width of 16, `--vcop_simd=16` should be used. This option controls the translation sequence calls to the VM. As an additional change to allow this option, the generated C source file will also define `VCOP_SIMD_WIDTH`.

Some kernels depend on a specific SIMD width and will not work correctly if extended to 16-way SIMD. Furthermore, increasing the SIMD factor may depend on certain properties of the data layout in memory. For example, image widths may be required to be multiples of 16 instead of 8. It is not possible for the migration tool to automatically detect these cases.

The following are examples of VCOP operations that cannot be trivially extended to 16-way SIMD.

- VBITPK – if kernel assumes results are 8-bit values
- VBITTR – assumes 8x8 transpose
- VBITUNPK – if bit mask (`src1[0]`) is assumed to be 8 bits
- Interleave/Deinterleave, including de-interleaving loads, interleaving stores, and vector operations – if kernel assumes interleaving on 8-lane boundaries. (If kernel avoids making assumptions about vector sizes or layouts, for example simply using deinterleave-on-read and interleave-on-write to improve throughput without regard to layout, then interleaving can be extended to wider SIMD widths.)
- Load with custom distribution – kernel C source format is tied to 8 lanes
- Load with expand – if kernel assumes 8-bit predicate
- Load with nbits – if kernel assumes 8-bit type for packed bit vector in memory
- Lookup table and histogram – the table layout in memory is tied to VCOP's 8-bank memory architecture, but certain cases of 16-way lookup and histogram are supported. See Section 5.4.8.

1.6 VCOP Virtual Machine

The VCOP virtual machine (VVM, or simply virtual machine) is a C++ library that implements VCOP operations on C7x. The C7x source code generated by the migration tool contains calls to the virtual machine. The virtual machine is intended to be an implementation mechanism; the programmer does not need to know anything about it or even be aware of its existence. However, it is also not intended to be a "black box." Programmers may find it useful to understand at least the API to the virtual machine so they can understand the code from the migration tool. Further, programmers may wish to consult the virtual machine to see specifically how VCOP operations are implemented for C7x.

The virtual machine consists entirely of inline functions and C++ classes. None of the classes are actually instantiated into objects; all the member functions are static inline, and the variables static. There is no object component of the virtual machine to include at link time; it consists entirely of header files.

The classes and functions use C++ templates to abstract low-level details between similar constructs. Template parameters specify things like data type, distribution mode, addressing mode, and SIMD factor; these items are determined by the migration tool and therefore known at compile time. The library uses template specialization to customize the implementation of a given operation for specific combinations of the template parameters.

The implementation uses the C7x low-level programming model, consisting of native vector (extended OpenCL) types and C7x intrinsics from `<c7x.h>`. That header file cross-references the intrinsic names with the actual C7x mnemonics; in this way the programmer can see what instructions are actually being generated.

A detailed description of the virtual machine is beyond the scope of this document. However, the core components of the VVM are as follows:

- vcop_arith.h – Defines the VCOP arithmetic operations.
- vcop.h – Contains core definitions for the VVM. (VCOP types, macros, etc.)
- vcop_lht.h – Defines operations that use the C7x lookup table. (`_LOOKUP`, `_HIST`, and some forms of `OFFSET_NP1` and `PDDA`.)
- vcop_load.h – Defines VCOP load operations, with the exception of forms in vcop_lht.h.
- vcop_store.h – Defines VCOP store operations, with the exception of forms in vcop_lht.h.
- vcop_vm.h – Central header. Contains shared templates used by other headers. Includes other core headers.

Kernel API

This chapter provides information about how the migration tool generates C7x functions that correspond to the EVE C-callable API.

Topic **Page**

2.1 Overview

A VCOP kernel written in VCOP Kernel-C is specified as C-callable function with arguments. The function body consists of one or more VCOP loop commands, table lookup commands, or histogram commands, collectively known as *vloop commands*.

On EVE, the kernel is invoked via a C-callable API consisting of five functions, which are generated by VCC. For a kernel named `kernel` written in VCOP Kernel-C, the API is:

- `unsigned int` **`kernel_init`**`(<args>, unsigned short pblock[])`
  Evaluate kernel-invariant expressions, including kernel arguments, into `pblock[]`
- `unsigned int` **`kernel_param_count`**`(void)`
  Return the size of the parameter block
- `void` **`kernel_vloops`**`(unsigned short pblock[])`
  Execute the kernel loops using parameters from pblock[]. This function contains the actual VCOP code.
- `void` **`kernel`**`(<args>)`
  Call `kernel_init()` and `kernel_vloops()` using a statically allocated parameter block
- `void` **`kernel_custom`**`()`
  Call `kernel_init()` and `kernel_vloops()` using a passed-in parameter block

The API gives the application varying levels of control over the kernel execution. A naive application can simply call the kernel as `kernel(<args>)`. A more sophisticated application that wants to control parameter buffer allocation and leverage double-buffering would call the lower level functions directly.

The migration tool generates these same five functions, as C++ functions for C7x. Therefore the functional interface to the kernel remains the same.

Figure 2-1 illustrates the kernel API in its original form for EVE and as translated for C7x.

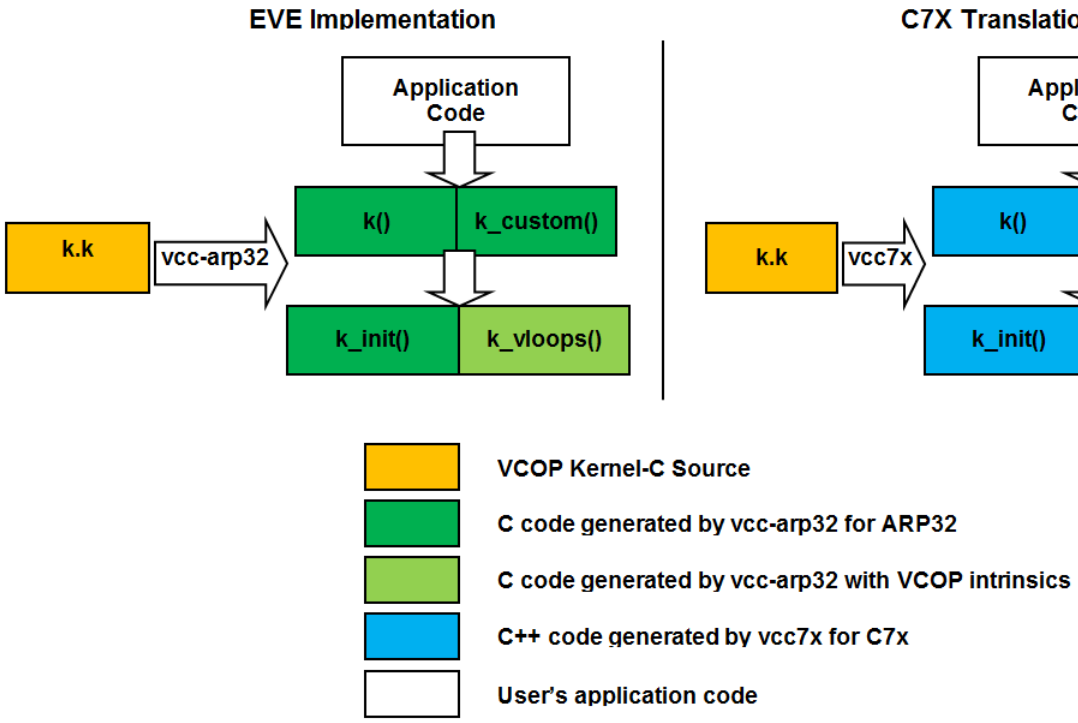

Figure 2-1. Kernel API

2.2 Parameter Block

The `init()` function generated as part of native kernel compilation for EVE computes kernel-invariant expressions (called *pexprs* in the Kernel-C language spec) and stores the computed values in a block of memory called the parameter block, or *pblock*. As the kernel executes, it references parameter values as memory-mapped 16-bit parameter registers (*pregs*) that are mapped to the location of the pblock. The parameter block is declared as an array of `unsigned short`, corresponding to the data type of the pregs. It may be allocated by the application code and passed to both the `init()` function and the `custom()` kernel function. Alternately, there is a statically allocated default variable called `<kernel>_pblock[]` that can be used when the application code does not need to control pblock allocation.

All parameterization of the `vloops()` function is done through the pblock. Arguments to the `kernel()` function are not passed to the `vloops()` function; instead they are captured by the `init()` function and accessed via the pblock.

2.2.1 Tvals Structure

To preserve the dispatch API, the C7x translation uses a similar approach. Expressions that are kernel invariant are precomputed by the `init()` function and stored in a memory block, then accessed by the translated kernel by passing the memory block's address to the `vloops()` function. However, the size, layout, and contents of the block is different. On C7x, restricting parameter elements to be 16-bit values is needlessly inefficient; parameter expressions often represent constants, addresses, or even vectors (for example Streaming Engine setup vectors). For this reason the pblock object on C7x is defined as a structure with fields defined by the migration tool. The address of the structure is passed to the `init()` function, which assigns to the fields of the structure, and the `vloops()` function, which accesses them. The structure has type `<kernel>_tvals_t`, and is referred to simply as the *tvals structure* throughout this document.

Even though the translated parameter block object is a structure, to preserve API compatibility, the pblock pointer parameters to the `init()` and `vloops()` function are declared as `unsigned short*`. The functions convert the pointer to point to the tvals structure by casting it.

The `param_count()` API function allows the client to allocate the pblock rather than using the built-in statically allocated one. It returns the required size of the pblock in `unsigned short` units. For example, a user-managed pblock might be allocated as:

```
typedef unsigned short ptype;
ptype *my_pblock = (ptype*)malloc(mykernel_param_count() * sizeof(ptype));
```

To preserve compatibility, for C7x the return value of the `param_count()` function is still in terms of `unsigned short` units, even though the underlying object is actually a structure. This allows client code that allocates the pblock as above to remain unchanged.

The tvals structure consists of one field for each kernel parameter to capture its value, plus a nested substructure for each vloop in the kernel to hold loop-specific expressions. Furthermore, each substructure is actually an array of substructures. This is to handle vloops that use the "repeat" feature, whereby the vloop runs multiple times using different parameter sets.

Table 2-1 illustrates the migration tool-generated tvals structure for a simple kernel with two vloop commands, the first of which is inside a repeat loop.

Table 2-1. Migration tool-generated parameter block access via tvals structure

Kernel-C Program

```
kernel(__vptr_int32 parm1, int N)
{
   // "repeat" loop
   foreach (roi, roi < N, 5)
   {
      // vloop 0
      for (...)
      {

      }
   }

   // vloop 1
   for (...)
   {

   }
}
```

Generated Tvals Structure

```
typedef struct
{
   __vptr_int32 parm1;   // kernel parm
   int N;                // kernel parm
   struct                // tvals for loop0
   {
      struct
      {
         int p0;
         uchar* p1;
         ...

      } tvals[5];
   } loop0;

   struct                // tvals for loop1
   {
      struct
      {
         uchar* p0;
         uint p1;
         uint16 p2;
         ...
      } tvals[1];
   } loop1;

} kernel_tvals_t;
```

Access to Parameters in vloops() Function

```
kernel_vloops(ushort *pblock)        // ushort* for API compatibility
{
   kernel_tvals_t *tvals = (kernel_tvals_t *)pblock;    // cast as tvals_t*

   // retrieve captured kernel parameters
   __vptr_int32 parm1 = tvals->parm1;
   int N = tvals->N;

   // access to kernel parameters
   ... parm1 ...

   // access to parameters for vloop 0 (index by repeat loop counter)
   ... tvals->loop0.tvals[roi].p0 ...

   // access to parameters for vloop 1
   ... tvals->loop1.tvals[0].p2 ...
}
```

Compatibility Warning: Pblock Size

The parameter block for C7x tends to be larger on C7x than VCOP. Programs which (dynamically or statically) allocate the pblock without using the kernel_pblock_size() function may fail if the allocated space is too small for the translated tvals structure.

2.2.2 Pblock Manipulation

Some VCOP applications have kernels that manipulate the parameter blocks of other kernels. This is to allow dynamic update of kernel parameters without the overhead of returning to ARP32 and re-calling the `init()` function. However, it is fragile in that it relies on the updating kernel to have direct knowledge of the pblock of the other kernel, manifested as hard-coded pblock offsets.

There is no direct way to support this for C7x translation since the pblock size and layout differ. Kernels that perform pblock manipulation will not work if directly translated for C7x.

Compatibility Warning: Pblock Manipulation
The size and layout of the pblock differs between C7x and VCOP. A kernel which tries to modify the pblock or otherwise depends on its contents will not execute correctly on C7x.

As an alternative means to support this functionality, the C7x migration tool adds an additional function to the dispatch API that allows for dynamic update of kernel parameters.

Recall that the `init()` function captures kernel parameters and stores them in the tvals structure. It also computes additional expressions used by the `vloops()` function and stores them in the tvals structure.

A new keyword, **__update**, identifies kernel parameters that may be dynamically updated. If any kernel parameters are declared using `__update`, the migration tool generates an additional function with the following signature:

`void` **`kernel_update`**`(`*`<update args>`*`, unsigned short pblock[])`

The arguments to the `update()` function consist only of the kernel parameters declared as `__update`. The function updates the pblock by re-capturing these parameters into the tvals structure, and re-evaluating any tvals that depend on them. The updated pblock can be passed to a re-invocation of the `vloops()` function.

Table 2-2 provides an example of a kernel that declares `__update` parameters, along with the generated code and the kernel dispatch code that calls it.

Table 2-2. Updating Kernel Parameters via update() API

Kernel-C Program

```
kernel(         __vptr_int32 parm1,      // non-updatable parameter
       __update __vptr_int32 parm2,      // updatable
                int          N)          // non-updatable
{
    ...
}
```

Generated Update Function

```
kernel_update(__vptr_int32 parm2,      // sig. includes only __update parameters
              ushort *pblock)
{
   kernel_tvals_t *tvals = (kernel_tvals_t *)pblock

   // recapture __update parameters
   tvals->parm2 = parm2;

   // re-compute tvals that depend on __update parameters
   tvals->loop0.tvals[0].p3 = ... parm2 ... ;

   ...
}
```

Client's Kernel Dispatch Code

```
kernel_init(buffer1, buffer2);        // (using built-in default pblock)
kernel_vloops();                 // first call
kernel_update(buffer3);             // updates pblock with parm2=buffer3
kernel_vloops();                 // second call
```

Loop Control

This chapter provides information about how the migration tool converts VCOP's vloop commands to C7x C code.

Topic **Page**

3.1 Overview

VCOP kernels consist of one or more vloop “commands”, each of which is expressed as a loop nest having up to 4 levels. The loops are controlled by a hardware looping mechanism. Logically, each loop is specified as a simple counter (loop control variable, or *LCV*) that starts at 0 and counts up by one. Trip counts are guaranteed to be loop invariant, although there is an early exit mechanism, discussed in Section 3.5. Each loop level is guaranteed to execute at least once. Trip counts are specified in the parameter block as 16-bit unsigned values, so the maximum number of iterations is 2^{16} or 65536.

For C7x, the migration tool simply generates loops using `for` expressions, just as they appear in the Kernel-C source, right down to using the same names for the LCVs. The migration tool adds `must_iterate` pragmas to indicate the built-in constraints on the trip counts.

3.2 Loop Control and Nested Loops

As generated, each loop level requires computation to control the loop: initialization of the LCV outside the loop, and an increment, compare, and branch inside the loop. The compiler and ISA provide a handful of mechanisms to reduce or eliminate this overhead.

First, the compiler typically converts simple counting loops to count down rather than up, slightly reducing the cost of the compare and branch operation.

Second, if the body of a loop consists entirely of another loop, the compiler can collapse the two loops into a single loop whose trip count is the product of the original loops.

Finally, and most significantly, the compiler can leverage an ISA feature called the Nested Loop Controller (NLC), which provides hardware loop control similar to VCOP for up to two levels of loop nesting.

The NLC allows loops to be collapsed even if the outer loop has code in addition to the inner loop, by providing automatically-generated predicates for the outer loop code.

The combination of the compiler’s collapsing and the NLC reduce loop control overhead to an acceptable degree. Virtually all kernels should have at least the two inner levels collapsed and controlled by the NLC. Often there are at least two levels that the compiler can collapse, so most loops execute as either singly nested loops, or in rare cases, doubly or triply nested loops. This is important because the C7x relies on software pipelining for performance, and only the inner loop can be pipelined.

3.3 Repeat Loops

The “repeat loop” feature of VCOP allows the programmer to wrap an outer loop around the loop nest of a vloops command. Each iteration of the outer loop executes with a new set of parameters from the pblock. A repeat loop is specified with the `foreach` construct in Kernel-C.

The translation of repeat loops uses a straightforward `for` loop. The tvals structure corresponding to the parameter block for a loop that uses the repeat feature is declared as an array, which is indexed by the repeat loop LCV to access parameter values.

3.4 Compound Conditions

VCOP supports conditional moves with compound predicates comprised of testing the LCVs for either the first or last iteration of each loop level. For example, the following kernel contains a conditional move that will execute only when all three loops are in their first iteration:

```
for(I0 = 0; I0 < N0; I0++) {
    for (I1 = 0; I1 < N1; I1++) {
      for (I2 = 0; I2 < N2; I2++)
      {
         ...
         // conditional move
         if (I0 == 0 && I1 == 0 && I2 == 0)
            Vx = Vy;
         ...
      }
```

The migration tool translates conditional moves as simple conditions (`if` statements), copying the condition verbatim from the Kernel-C expression. This will prevent additional optimizations such as collapsing and use of the NLC because it requires the LCVs to be explicitly instantiated.

3.5 Early Exit

VCOP has a VEXITNZ instruction which conditionally breaks out of a given loop nest or repeat loop. The condition is based on comparing lane 0 of a vector register to zero. In Kernel-C, early exit is expressed as a conditional `goto` statement and a label, for example:

```
__vector acc1, acc2, flag;

for (int I1 = 0; I1 < 1; I1++)
{
    for (int I4 = 0; I4 < num_pix[I0]; ++I4)
    {
       flag = (acc1 > acc2);
       if (flag[0]) goto end_loop1;
    }
}
end_loop1:;
```

For C7x translation, the migration tool simply copies the goto and label statements into the generated output verbatim and leaves it to the C7x compiler to generate the appropriate control flow. A loop with an early exit cannot use the NLC, but it may be software pipelined as a so-called “irregular loop”.

Addressing

This chapter provides information about how the migration tool converts VCOP's address generators to C7x emulation mechanisms.

Topic **Page**

4.1 Overview

VCOP has dedicated address generators (*Agens*) that facilitate efficient multidimensional access patterns for loads and stores in the loop. VCOP's Agens compute address offsets; individual loads and stores supply base addresses from the parameter block which are combined with the offsets to form the effective address for the load or store. There are eight Agens, but a single Agen can be shared among different loads or stores with different base addresses.

The C7x has three different mechanisms for emulating VCOP's Agens. They are, from most efficient to least efficient: the Streaming Engine (SE), the Streaming Address Generator (SA), and regular indexed addressing.

4.2 Streaming Engines

The SEs provide the most efficient addressing, but are restricted as follows:

- There are only two of them.
- They can only be used for loads (not stores).
- The base address is pre-initialized by the SE_OPEN step; therefore a given SE cannot be shared between multiple loads with different base addresses.

The migration tool allocates the two SE resources to what it considers to be the two highest priority loads. Loads in the innermost loop are considered to have the highest priority. The heuristic simply picks the first two loads in the innermost loop; if there are fewer than two, it moves to the next outer loop.

For SE-based loads, the migration tool generates the following sequence of steps:

1. In the `init()` function, the migration tool generates a call to the `SE_init()` template function in the virtual machine, which returns an SE setup vector. (The ISA spec refers to the setup vector as an SE template; here we use the term *setup vector* to avoid confusion with the virtual machine's C++ templates). The setup vector is saved in the tvals structure for later access by the `vloops()` function. The setup vector consists of static (compile-time) and dynamic (run-time) values. The static values correspond to flags in the SE setup vector and are determined from the distribution mode and data type. These are passed as template parameters to `SE_init()`. The dynamic values correspond to stride and trip count values that are determined from the terms in the Agen expression and loop trip counts. These are passed as runtime arguments to `SE_init()`.
2. Also in the `init()` function, the migration tool generates the expression that represents the base address and saves that in another field of the tvals structure.
3. In the `vloops()` function, outside the outermost loop, the migration tool generates a call to the `SE_OPEN()` intrinsic, passing it both the setup vector and the base address from the tvals structure.
4. The load instruction in the loop simply uses an `__se_ac_<type>` intrinsic for the access, which turns into a quasi-register operand `SEn++` containing the loaded value.
5. As an optimization, the compiler may copy-propagate the `SEn++` operand into the instruction where the value is used, thereby eliminating the load instruction altogether.
6. The migration tool generates a call to the `SE_CLOSE()` intrinsic after the loop nest.

4.3 Streaming Address Generators

C7x has four streaming address generators (SA) that provide zero-overhead address offset calculations for nested loops, similar to VCOP's Agens. The SAs are similar to the SEs, with the following differences:

- They can be used for both loads and stores.
- They do not offer the same pre-fetch advantage as the SEs.
- They do not have the flexible formatting options that allow seamless emulation of VCOP's distribution modes.
- The base address is not pre-initialized; therefore SAs can be shared between loads and stores that use the same Agen on VCOP.

After assigning the two SEs to the highest priority loads to SEs, the migration tool assigns the four SAs to the next highest priority loads and stores (perhaps more than four if some share Agens). The process for using SAs is similar to that for SEs in the previous section. The primary difference is that the base address is passed to the access intrinsic rather to the `SA_OPEN()` call.

4.4 Indexed Addressing

Any loads or stores that remain after exhausting the SE or SA resources are generated using normal indexed addressing. The migration tool defines a variable corresponding to the Agen whose type is `__agen`, which is a typedef for `int`. The agen variable represents the address offset in bytes.

An access with base address `b` and agen `a` is generated as the C expression `*(<type>*)((char*)b + a)`. This generally results in no overhead for the access itself, as the compiler generally keeps both `b` and `a` in registers for the duration of the loop nest, so the expression compiles to a simple indirect operand such as *Rega[Regb].

The overhead results from having to update the agen as the loops iterate. These updates generally involve adding a constant term at each loop level corresponding to the stride at that level. For levels outside the innermost level, the stride is adjusted so as to rewind the inner level. The agen adjustment values are computed in the `init()` function from the coefficients in the Agen expressions and the trip counts, and stored in the tvals structure.

For example, the following code shows the addressing operations generated for a loop translated using indexed addressing:

```
__agen A0, A1, A2;     // typedef int __agen

for (I1 ... )
{
   for (I2 ...)
   {
      for (I3 ...)
      {
         for (I4...)
         {
            Vreg0 = *(tvals->p4 + A0);         // load using A0
            Vreg1 = *(tvals->p5 + A1);        // load using A1

            A0 += 2;           // update A0
            A1 += 2;           // update A1
         }
         A0 += tvals->p8;            // outer loop updates
         A1 += tvals->p11;
      }
      *(tvals->p7 + A2) = Vdst;    // store using A2

      A0 += tvals->p9;              // outer loop updates
      A1 += tvals->p12;
      A2 += 16;
   }
   A0 += tvals->p10;                // outer loop updates
   A2 += tvals->p13;
}
```

The agen updates are generated as statements of the form `a += tvals->pN`, positioned at the end of the loop at the appropriate level. These generally turn into a single ADD instruction, but since they appear in outer loops, can hamper loop collapsing. For loops collapsed with NLC, the NLC-supplied predicate can be used to predicate any agen adjustments in the outer loop. The total overhead of the agen in this case is generally two instructions: the instruction to fetch the predicate from the NLC, and the (predicated) add instruction to update the agen. For loops not collapsed with NLC, the overhead is higher, because the presence of the agen update in the outer loop usually prevents collapsing, necessitating explicit loop control for that loop level.

4.5 Circular Addressing

VCOP has a hardware feature that supports circular addressing for loads and stores. Addresses consisting of a base and offset computed such that the offset is taken modulo a fixed buffer size, causing the address to wrap back to the beginning of the buffer as the offset increases. In VCOP Kernel-C circular addressing is specified using a modulo (%) operator in the address calculation.

C7X does not directly support circular addressing using normal indirect addressing modes. The Streaming Engine supports circular addressing, but only for loads.

The migration tool implements circular addressing using an explicit computation at the point of the access. The effective address is computed as follows:

```
unsigned long mask = BUF_SIZE - 1;
void *addr = (base & ~mask) | ((base + offset) & mask);
```

The mask is invariant and can be pre-computed outside the loop. The rest of the computation requires 4 instructions: an ADD, two ANDs, and an OR.

Operations

This chapter describes the translation of load operations (see Section 5.1), store operations (see Section 5.2), arithmetic operations (see Section 5.3), and lookup and histogram table operations (see Section 5.4).

Topic **Page**

5.1 Load Operations

VCOP's fundamental load operation fetches a vector of 8 consecutive elements from memory into the 8 lanes of a vector register. In memory the elements are 8, 16, or 32 bit signed or unsigned values; they are sign or zero extended into the 40-bit lanes of the register. On C7x the migration tool models VCOP's 40 bit lanes as 32 bit lanes; a vector register contains 16 of these lanes. (In 8-way SIMD mode only 8 of these 16 lanes are used; in 16-way SIMD mode all 16 are used.)

VCOP has various additional "distribution modes" that provide for alternate data layouts in memory: for example, as the data is read in it may be oversampled (read each element multiple times), undersampled (skip elements), deinterleaved, and so on. Descriptions of the distribution modes can be found in the VCOP CPU manual or the programmer's guide.

Thus, load operations are characterized primarily by the element type in memory and distribution mode. The virtual machine has a template class called `vcop_load` that implements the various combinations using C7x operations. The template parameters of the class specify the type and distribution mode, along with a specification of what kind of low-level addressing to use (SE, SA, or indirect) and the number of SIMD lanes to emulate (8 or 16). The class has two methods, `load()` and `SE_load()`, which implement the load operation as specified by the template parameters. The `load()` method implements non-SE-based loads, and the `SE_load()` method implements SE-based loads.

For example, here is the translation of a VCOP load instruction using the CIRC2 distribution mode and SA-based addressing.

Kernel-C:

```
__vptr_uint16 in;
Vreg = in[Agen0].circ2();
```

translates to:

```
vcop_load<ushort, circ2, sa1adv>::load(Vreg, (uchar*)(tvals->p1));
```

The template parameters `ushort`, and `circ2` specify the data type and distribution mode. The template parameter `sa1adv` tells the template to use SA-based addressing, with SA1 and advancing enabled. The SIMD factor defaults to 8. The runtime argument `Vreg` is the destination vector register (passed by reference, since the load writes into it). The `tvals->p1` expression is the base address. The `load()` method, when load template is specialized for `ushort`, `circ2`, and `sa1adv`, results in generation of the specific C7x sequence to implement that combination.

The specific C7x instruction sequences generated by specialized `load()` methods that result from all combinations of type and distribution mode can be determined by examining the template specializations in the header files.

The `load()` methods need to handle loading the vector elements according to the distribution mode, and sign- or zero-extension. Most combinations of type and distribution mode are covered by a single C7x instruction. A few need additional instructions to exactly mimic VCOP's specific modes.

In general SE-based loads, invoked via the `SE_load()` method, simply rely on the SE configuration as setup in the `init()` function (see Section 1.4), and translate to a simple access of the corresponding SE source register.

The basic setup for the SE is based on the data type. For example, signed 16-bit data uses element type __SE_ELETYPE_16BIT and __SE_PROMOTE_2X_SIGNEXT for sign-extension to 32 bits. The default vector length for 8-way SIMD is __SE_VECLEN_32BYTE, for 8 lanes of 32-bit (4-byte) data. Additional SE features used to implement the specific distribution modes are configured by the various specializations of the `vcop_load` template.

C7x can speculatively load without risk of faulting. Therefore most of the load sequences simply load a full C7x vector's worth of data – that is, 16 lanes of 32-bit values. In 8-way SIMD mode, the extra values are simply unused.

5.2 Store Operations

Similarly to loads, VCOP's store operations are characterized by data type and distribution mode. In addition to distribution mode, there are two primary considerations for translation of VCOP stores for C7x: packing, and lane masking.

Packing is the opposite of sign extension. The source data in C7x registers is 32 bits wide. When storing to 16- or 8-bit element types, the elements must be truncated to that size. The C7x has direct instruction support for such packing stores. Packing depends on size but not signedness. Signed and unsigned types generally use the exact same sequence, so there are three fundamental translation sequences for each mode, corresponding to 8-bit, 16-bit, or 32-bit data.

Signedness does come into play for rounding or saturation; these are covered in Section 5.2.3 and Section 5.2.4.

Unlike loads, translation of stores is sensitive to the number of lanes. While excess lanes can be safely loaded and ignored, stores must take care to only store the number of lanes being modeled. That means in the default 8-way SIMD mode, stores are limited to storing only 8 lanes, even though the C7x vectors contain 16 lanes of (32-bit) data. There is instructional support for such partial-vector stores for some cases, but not all. In particular there are no partial-vector packing stores. Thus packing stores in 8-way SIMD mode that use regular indirect addressing require the use of an explicit predicate to mask the store of unused lanes. This lane-masking predicate is constant and can be computed outside the loop.

When using SA-based addressing, the SA automatically provides lane masking based on the VECLEN flag in the SA setup vector. In this case the explicit lane masks are not needed.

The virtual machine has a template class called `vcop_store` that implements the various combinations in terms of C7x operations. The template parameters of the class specify the type and distribution mode, along with a specification of what kind of low-level addressing to use (SA or indirect), and the number of SIMD lanes to emulate (8 or 16). The class has two methods: `store()` for regular stores and `store_pred()` for predicated stores. For example, here is the translation of a store instruction using the DS2 distribution mode and SA-based addressing.

Kernel-C source:

```
__vptr_s8 out;
out[Agen1].ds2() = Vreg;
```

translates to:

```
vcop_store<char, ds2, sa0adv>::store(Vreg, (uchar *)(tvals->p2));
```

The template parameters `char`, and `ds2` specify the data type and distribution mode. The template parameter `sa0adv` tells the template to use SA-based addressing, with SA0 and advancing enabled. The SIMD factor defaults to 8. The runtime argument `Vreg` is the source vector register. The `tvals->p2` expression is the base address.

Most cases of unconditional stores using the basic store distributions modes translate to a one- or two-instruction sequence. Collating stores store a data-dependent number of elements in packed fashion and are therefore more involved.

Efficiency Warning: Collating Stores
Collating stores require a long translation sequence and may perform poorly.

5.2.1 Predicated Stores

For some distribution modes, VCOP supports explicit predication. A vector register can be specified as a predicate. If a lane of the predicate is zero, the corresponding lane of the source vector is blocked from being stored.

On C7x predicates are represented as bit vectors with one bit corresponding to each byte of the source register. To emulate VCOP's predicated stores, the C7x predicate must be generated by explicitly comparing the register containing the predicate vector to zero. This generates a word-wise predicate corresponding to the 32-bit words of the predicate vector.

Furthermore, if the store operation requires an explicit lane mask, the computed predicate must be combined with a lane-mask predicate using an AND instruction. Explicit lane masks are required when not using SA-based addressing, since the Streaming Address Generator provides built-in lane masking.

5.2.2 Scatter and Transposing Stores

VCOP has three distribution modes which store vector elements in a non-linear fashion. The OFFSET_NP1 mode stores each element 9 words apart, effecting a 9x8 transpose. (The 9-word offset causes each element to be stored into a different memory bank, to achieve a parallel store). The SDDA (`s_scatter` in Kernel-C) mode stores each element at a data-dependent offset; the offsets come from another vector. The PDDA (`p_scatter` in Kernel-C) mode is like SDDA except the offsets must be such that each element goes to a different bank, so they can be parallel. The PDDA mode is also typically used for transpose.

There is no direct support for these operations on C7x. On C7x, transpose is supported by the Streaming Engine, but only for loads and not for stores.

For the specific case of OFFSET_NP1 used to store into a scratch buffer for a transpose operation, the migration tool may detect this and re-arrange the layout of the scratch buffer so as to implement the transpose using the streaming engine. See Section 5.2.3.

Otherwise, the migration tool generates naive sequences for these modes, storing elements one at a time rather than in parallel.

Efficiency Warning: Parallel Stores
Unless the above optimizations are enabled, kernels that use these distribution modes will have inefficient translations.

5.2.3 Optimization of OFFSET_NP1-Based Transpose

VCC may recognize transpose sequences that use OFFSET_NP1. The general pattern is to store into a scratch buffer using OFFSET_NP1, then read the scratch buffer back using NPT loads. Unfortunately there is no direct translation for the OFFSET_NP1 store. However, the streaming engine does support a transposed read mode. If transpose recognition is enabled, the migration tool *may* transform the sequence to use non-transposed stores instead of OFFSET_NP1 stores, and transposed loads using the streaming engine instead of normal vector loads. Thus, the transpose operation shifts from the store to the subsequent load. The layout of the data in the scratch buffer is altered with respect to its VCOP layout, so this transformation only works when the scratch buffer is used only for the transpose operation and not otherwise used.

Transpose detection and transformation may be enabled by:

1. Enabling the `--transpose` command line option. This option enables automatic detection of transpose sequences. This will apply transpose at every possible point in the file. If a kernel that should use transpose is in the same file as one that shouldn't, they should be separated into two files.
2. Applying the `__tscratch` keyword to a parameter. (For example, `__tscratch __vptr_uint32 scratch_buffer`) This method of enabling transpose will take effect even if `--transpose` is not specified.

The transpose transformation may be performed under the following conditions:

- All stores in the sequence are OFFSET_NP1.
- All loads in the sequence are NPT.
- The buffer used for the transpose is not later examined by the caller. (The data in the buffer will not be transposed.)
- Enough streaming engines and streaming address generators are available.
- The stores/loads are 32 bit. (The streaming engine only supports all transpose read configurations for 32 bit data. If the only reason a transpose is not performed is the data type, VCC will warn.)

The transpose transformation will correctly handle unrolled reads or unrolled writes and transform them as a set. In addition, the transpose transformation will correctly handle a transpose scratch buffer that has been split such that the one portion is used separately from another portion. However, the transpose transformation will not correctly handle a combination of unrolled reads/writes AND a transpose scratch buffer that has been split. (It becomes impossible for VCC to disambiguate the offset for the unroll from the offset for the split.)

5.2.4 Rounding Stores

VCOP can perform rounding or truncation in conjunction with stores. The migration tool treats these as discrete operations and generates them prior to the store.

Rounding is translated as if it were an arithmetic operation. That is, a rounding store of the form:

```
out[Agen] = Vsrc.round(N); // round at Nth bit
```

is translated as if were expressed as:

```
Vbits = N;
Vtemp = Vsrc.round(Vbits); // VSHRRW
out[Agen] = Vtemp;
```

Rounding adds one instruction in addition to the normal store instruction or sequence.

Similarly, truncation is translated as a discrete right shift. A truncating store of the form:

```
out[Agen] = Vreg.truncate(N);
```

is translated as if it were:

```
Vbits = -N; // negate so shift goes right
Vtemp = Vsrc << Vbits; // actually >>
out[Agen] = Vtemp;
```

This also results in one additional instruction.

5.2.5 Saturating Stores

Like rounding, a VCOP store that includes saturation is translated as if were two operations: an explicit saturation operation, followed by the store. The saturation operation operates on 32-bit elements regardless of the data type of the store. VCOP supports several forms of saturation: SYMM, ASYMM, 4PARAM, and so on. Fundamentally, all of the forms operate as follows:

```
saturate(min, minset, max, maxset) = (x < min) ? minset
                                   : (x > max) ? maxset : x;
```

In the typical case when the saturation bounds are the same as the min/max set-to values, that is `min == minset` and `max == maxset`, the C7x translation is an efficient two instruction sequence:

```
VMINW       Vsrc,Vmax,Vdst
VMAXW       Vdst,Vmin,Vdst
```

If the saturation bounds are to a power of two boundary, such as from 0 to 255, a single saturation instruction will be used:

```
VGSATUW     Vsrc,Cwidth,Vdst
```

If the saturation bounds are different from the min/max set-to values, a less efficient 4-instruction sequence is required:

```
VCMPGTW      Vmin,Vsrc,Pred0
VSEL         Pred0,Vminset,Vsrc
VCMPGTW      Vmax,Vsrc,Pred1
VSEL         Pred1,Vsrc,Vmaxset
```

For unsigned vectors (see Section 1.4), unsigned forms of the compares are used.

C7x has a dedicated instruction for saturating to the range of a signed 16-bit value: VSATWH. Therefore the following statement:

```
__vptr_s16 out;
out[Agen] = Vsrc.saturate(); // saturates to (-32768, 32767)
```

translates to a single instruction.

VCC removes saturations that it determines to have no effect.

5.3 Arithmetic Operations

Most of VCOP's arithmetic operations translate directly to either one C7x instruction or a short sequence.

VCOP arithmetic operations generally operate on vectors of signed 40-bit elements. Some VCOP operations ignore the upper 8 bits and operate on 32-bit elements. C7x lacks direct support for 40-bit arithmetic. One option is to model each lane as a 64-bit element but in practice most kernels rely only on 32 bits of precision. So the migration tool models VCOP vectors as having 32-bit elements and translates them accordingly (see Section 1.4).

Most arithmetic operations are sign-agnostic, but in some cases treating elements as unsigned rather than signed can correct translation errors arising from the loss of the guard (see Section 1.4). Therefore some translations have unsigned forms in addition to the default signed forms.

In cases where an operation is represented by a C operator, the migration tool generates the expression using the operator, and the C7x compiler generates the appropriate vector instruction. For example, for this VCOP Kernel-C statement:

```
Vdst = Vsrc1 + Vsrc2;
```

The migration tool simply copies the expression verbatim and the compiler generates a VADDW instruction.

In other cases where an operation translates directly to a C7x instruction, the compiler generates a C7x intrinsic to invoke the instruction. For example:

```
Vdst = min(Vsrc1, Vsrc2);
```

translates to

```
Vdst = __min(Vsrc1, Vsrc2);
```

which turns directly into a VMINW instruction.

If there is no direct translation to a single instruction, the migration tool relies on the virtual machine to provide the translation. Each operation that does not correspond to a C operator is implemented by a class in the virtual machine. Like the load and store classes, these are template classes so that variations can be specified at compile time by template parameters. For example, this VCOP Kernel-C statement:

```
Vdst = unpack(Vsrc1, Vsrc2);
```

translates to:

```
bit_unpack<int>::apply(Vdst, Vsrc1, Vsrc2);
```

The element type is specified by the template parameter `int`. All arithmetic classes have a single method called `apply()` that implements the translation. In this case the `apply()` method invokes a sequence of four C7x intrinsics..

5.3.1 Vector Compares

Vector compare operations map directly to C7x's vector compare instructions, but require an additional VSEL instruction to turn the vector predicate generated by the compare into a vector of 1's and 0's in order to mimic VCOP.

5.3.2 Multiplication with Rounding, Truncation, or Left Shift

VCOP multiplication operations (VMPY, VMADD, and VMSUB) support optional scaling as part of the operation. The scaling can take the form of rounding, truncation (right shift), or left shift.

As with stores (see Section 5.2.3), for translation these are treated as distinct operations. Each adds an additional instruction to the normal translation sequence for these operations: a single vector round or shift instruction applied to the result operand of the VMPYW instruction.

5.4 Lookup and Histogram Table Operations

VCOP has dedicated support for Lookup and Histogram Table (*LHT*) operations that access memory in a non-linear fashion. Both operate by indexing into a table using index values that contain offsets from the table base. In the case of lookup table, the lookup operation loads from memory at the indexed locations and returns the values. In the case of histogram, the update operation increments the memory at the indexed locations.

Both lookup table and histogram support the notion of parallel tables, in which up to 8 distinct tables are referenced using index values contained in the lanes of a vector register (and in the case of lookup, values read from the table are returned in corresponding lanes). To enable parallel access to the tables, the tables are interleaved in memory such that each distinct table occupies a different memory bank or banks than the others.

C7x supports LHT operations in a very similar way. However there are two important differences that fundamentally affect the translation of these operations to C7x.

First, VCOP supports LHT operations directly on its local memory, consisting of the IBUFA, IBUFB, and WBUF buffers. Each is 32KB, for a total of 96KB. Typically these buffers will be mapped to L2 on C7x. But C7x supports LHT operations only on a 32KB area of L1D. Therefore LHT operations require table data to be copied from L2 into L1D before the operation, and then (in the case of histogram) copied back out.

Second, VCOP's memory is laid out as 8 banks, each 32 bits wide. Each full-width line from the memory interface contains 8x32 or 256 bits. C7x's memory is laid out as 16 banks, each 64 bits wide. Each line contains 16x64 or 1024 bits. For LHT operations that use parallel tables, the bank geometry affects the table layout in memory.

For example, consider a LHT operation with 4 parallel tables and 32-bit elements. We'll call them T0, T1, T2, and T3. (In the following description, for simplicity, "locations" refer to word offsets within the linear address space of table memory. In reality, both machines are byte addressable.) On VCOP each table occupies 2 of the 8 banks. In each table line, the two banks corresponding to a given table contain 2 elements: locations 0 and 1 are part of table T0; locations 2 and 3 are part of T1, and so on. On C7x, each table occupies 4 of the 16 banks; in each line the 4 64-bit banks contain 8 table elements. Locations 0—7 are part of T0; locations 8—15 are part of T1, and so on. Figure 5-1 illustrates the two layouts for the table in this example. In this figure, TN[M] represents the element at index M of table N. Shading indicates memory banks. Heavy borders indicate table boundaries.

VCOP Table Layout for 4 Parallel Tables, 32-bit Elements

	Word Addr								
Line0, Banks 0-7	0x0	T0[0]	T0[1]	T1[0]	T1[1]	T2[0]	T2[1]	T3[0]	T3[1]
Line1, Banks 0-7	0x8	T0[2]	T0[3]	T1[2]	T1[3]	T2[2]	T2[3]	T3[2]	T3[3]
Line2, Banks 0-7	0x10	T0[4]	T0[5]	T1[4]	T1[5]	T2[4]	T2[5]	T3[4]	T3[5]
Line3, Banks 0-7	0x18	T0[6]	T0[7]	T1[6]	T1[7]	T2[6]	T2[7]	T3[6]	T3[7]
Line4, Banks 0-7	0x20	T0[8]	T0[9]	T1[8]	T1[9]	T2[8]	T2[9]	T3[8]	T3[9]
Line5, Banks 0-7	0x28	T0[10]	T0[11]	T1[10]	T1[11]	T2[10]	T2[11]	T3[10]	T3[11]
Line6, Banks 0-7	0x30	T0[12]	T0[13]	T1[12]	T1[13]	T2[12]	T2[13]	T3[12]	T3[13]
Line7, Banks 0-7	0x38	T0[14]	T0[15]	T1[14]	T1[15]	T2[14]	T2[15]	T3[14]	T3[15]

C7x Table Layout for 4 Parallel Tables, 32-bit Elements

	Word Addr								
Line0, Banks 0-3	0x0	T0[0]	T0[1]	T0[2]	T0[3]	T0[4]	T0[5]	T0[6]	T0[7]
Line0, Banks 4-7	0x8	T1[0]	T1[1]	T1[2]	T1[3]	T1[4]	T1[5]	T1[6]	T1[7]
Line0, Banks 8-11	0x10	T2[0]	T2[1]	T2[2]	T2[3]	T2[4]	T2[5]	T2[6]	T2[7]
Line0, Banks 12-15	0x18	T3[0]	T3[1]	T3[2]	T3[3]	T3[4]	T3[5]	T3[6]	T3[7]
Line1, Banks 0-3	0x20	T0[8]	T0[9]	T0[10]	T0[11]	T0[12]	T0[13]	T0[14]	T0[15]
Line1, Banks 4-7	0x28	T1[8]	T1[9]	T1[10]	T1[11]	T1[12]	T1[13]	T1[14]	T1[15]
Line1, Banks 8-11	0x30	T2[8]	T2[9]	T2[10]	T2[11]	T2[12]	T2[13]	T2[14]	T2[15]
Line1, Banks 12-15	0x38	T3[8]	T3[9]	T3[10]	T3[11]	T3[12]	T3[13]	T3[14]	T3[15]

Figure 5-1. Differences in Table Layout Between VCOP and C7x

VCOP kernels assume tables are laid out according to VCOP's geometry; therefore this layout must be preserved at kernel boundaries. In order to operate on such tables with C7x LHT operations, the tables must be rearranged to conform to C7X's geometry.

In general, a kernel that uses LHT operations must go through the following steps when translated to C7x:

1. (In the `init()` function) Compute the LTCR control register flags to configure the table for the number of parallel tables and element type, and store the configuration in the tvals structure.
2. (Before loop)
    1. Copy table data from L2 location to L1D.
    2. Rearrange table layout from VCOP to C7x.
3. (Before loop) Configure the table using the value from the tvals structure
4. (Within loop) Perform the lookup or histogram operation on table in L1D
5. (After loop, histogram only)
    1. Copy table data from L1D to L2
    2. Rearrange table layout from C7x to VCOP

In practice, steps 2a and 2b (copy and rearrange) are combined: the table is rearranged on-the-fly as it is copied. Similarly, steps 5a and 5b are combined.

Efficiency Warning: LHT Operations
The translation of LHT operations suffers from the overhead introduced by copying and rearranging the table data. This step takes approximately one cycle per 512 bits (64 bytes) of data. The exact penalty depends on the size of the table.

The virtual machine contains two template classes, `vcop_lookup` and `vcop_hist`. These implement the translation steps in the previous outline. The template parameters specify the element data type, the number of parallel tables, and in the case of `vcop_lookup`, the interpolation degree. Both classes are based on another template called `LHT_base` which provides functionality common to both classes.

5.4.1 Determination of Table Size

In order to generate the copy-in/copy-out operations, the migration tool needs to know how large the table is. Unfortunately there is nothing in the Kernel-C source code that definitively indicates the size of a lookup or histogram table (the tables are referred to via a pointer to their base address). However, the migration tool can conservatively estimate the maximum required size by analyzing the possible range of values of the index vector used to index the table.

As a starting point, the absolute maximum table size is 32K bytes. That's the size of each VCOP memory block, and the maximum size reservable for table data in L1D on C7x.

On VCOP, the index vector can only be defined by loading it from memory. If the load's data type is `__vptr_int8` or `__vptr_uint8`, the maximum index value is either 127 or 255 respectively.

Alternatively, saturation is often applied to the index values so as to insure the table access is within its bounds. The migration tool uses the max-set value from the saturation expression to determine an upper bound on the index value and therefore the table's size.

To indicate the exact size of a table in bytes, the `_HISTOGRAM` and `_LOOKUP` directives have a new optional parameter, `table_size`:

```
_LOOKUP(num_tables, num_pts, duplication, table_size)
_HISTOGRAM(num_copies, duplication, table_size)
```

(Duplication is covered in Section 5.4.8) If the table size is not specified using the directives, and cannot be inferred by saturation or data size, the maximum size of 32K bytes will be used. The migration tool will issue a warning in this case.

5.4.2 Table Configuration

On C7x, LHT operations are controlled by a handful of control registers. C7x supports up to 4 table configurations that can be simultaneously active. On VCOP only one table operation can be active at a time so the migration tool always uses configuration 0 for the LHT operation. This configuration is enabled by writing `__LUT_ENABLE_0` into the LTER register and configured via the LTBR0 and LTCR0 control registers. The LTBR specifies the base of the configured table in L1D. Since there is only one active table the migration tool always establishes the base address at 0, and allocates all 32K bytes for configuration 0.

The LTCR configuration specifies table properties such as number of parallel tables, element size and signedness, element promotion, and saturation behavior (for histogram). The migration tool configures these fields according to the properties of the VCOP table. Elements are always promoted to 32-bits since that's how translated vectors are modeled.

The table configuration is computed by the `LHT_base::config()` method. The migration tool generates a call to this method in the kernel `init()` function and stores the config value in the tvals structure. Then, in the `vloops()` function, the migration tool generates a call to `LHT_base::open()`, which copies the config value from the tvals structure into LTCR0, thereby configuring the table.

5.4.3 Copy-in Operation

The copy-in operation is responsible for copying table data from its “permanent” location in L2 into L1D so that an LHT operation can be performed. The source table in L2 is in VCOP layout; the destination table in L1D is in C7x layout.

This operation is performed by the `LHT_copy_in::copy_table_in()` method of the virtual machine. A pointer to the table in L2 and its size in bytes are passed as parameters. The size is rounded up to a multiple of 128 bytes (1024 bits), which is the line size of banked tables on C7x.

The table is read and written one 1024-bit line at a time. Each line is read from L2 as a pair of 512-bit vectors using the Streaming Engine. The pair of vectors, containing four VCOP lines, are rearranged using two VPERM instructions. Then, they are written into L1D using a LUTINIT instruction.

Thus each 1024-bit chunk requires 2 SE-based vector loads, two VPERMs, and two LUTINITs. The resultant loop pipelines at an ii (initiation interval) of 2. The throughput is 512 bits per cycle.

In order to use LUTINIT to populate a table in L1D, the table must be configured as one parallel table, allowing the lanes of the payload vectors to be written into the table in linear fashion. This is regardless of how the table is configured for the LHT operation itself. So there is an independent LTCR configuration that applies only to the copy-in operation. This configuration is computed during the `init()` function by a call to the `copy_in_config()` method, and stored in the tvals structure.

Similarly, the SE configuration used for the copy-in operation is computed during `init()` by the `copy_in_SE_config()` method and stored in the tvals structure.

5.4.4 Copy-out Operation

The copy-out operation is essentially the reverse of the copy-in operation. The operation is performed by the `LHT_copy_out::copy_table_out()` method, which is called after a loop containing a histogram operation. The table will also be copied out after a qualifying OFFSET_NP1 or PDDA operation. (See)

Like copy-in, the table is read and written one 1024-bit line at a time. Each line is read from L1D as a pair of 512-bit vectors using two LUTRD (lookup read) instructions, rearranged using two VPERM instructions, and written using normal vector stores. The loop pipelines at an ii of 2, resulting in a throughput of 512 bits per cycle.

For purposes of the LUTRD used for the copy-out, the table is considered to be 16 parallel tables containing 32-bit elements. This enables each LUTRD instruction to read the maximal payload of 512 bits. The copy-out LTCR configuration is computed during the `init()` function by a call to the `copy_out_config()` method and stored in the tvals structure.

5.4.5 Index Adjustment from Non-zero Agen

The general form of a lookup table operation in VCOP Kernel-C is:

```
Vdst = base[Agen].lookup(Vindex);
```

The Agen is added to the base address of the table before adding the index offsets. In typical usage, the Agen remains fixed at zero, and no adjustment is needed. However some kernels may use non-zero or varying Agens as an additional addend to vary the table base dynamically. To account for this, the migration tool adds the offset from the Agen to each lane of the index vector prior to indexing the table. The adjustment calculation is:

```
Sadj = ((Agen / Ntables) / Elemsz);     // scale bytes -> table index
Vindex_adj = Vindex + Sadj;             // add to each lane vector
```

The division by `Ntables` accounts for the fact that when there are multiple parallel tables, the Agen represents an offset into the combined table data, not an offset into each table. The division by `Elemsz` accounts for the fact that the Agen is a byte offset, while the index values are element offsets. Both are compile-time constants so the two divisions typically simplify to a single shift instruction. Furthermore, if the Agen does not vary inside the loop, the computation of `Sadj` can be hoisted outside the loop.

In summary, if the Agen used in a LHT operation varies inside the loop, two additional instructions are required in the loop (shift and add).

5.4.6 Lookup Operation

Once the table is properly configured, copied to L1D, and arranged for C7x layout, the lookup operation itself is straightforward. This VCOP Kernel-C statement:

```
Vdst = base[Agen].lookup(Vindex);
```

translates to:

```
LUTRD LTBR0,Vindex_adj
```

where LTCR0 specifies the table configuration and `Vindex_adj` represents the adjusted index vector as described in Section 5.4.5.

The table configuration handles promotion of the table elements to 32-bits and interpolation (where multiple consecutive elements are returned for each index).

The lookup operation is implemented by the `vcop_lookup::lookup()` method of the virtual machine.

5.4.7 Histogram Update Operation

The histogram update operation is very similar to the lookup operation. The statement:

```
Vreg = base[agen].hist_update(Vindex, Vweight);
```

translates to

```
WHIST Vindex_adj, LTBR0, Vweight
```

where LTCR0 specifies the table configuration, `Vindex_adj` represents the adjusted index vector as described in Section 5.4.5, and `Vweight` is the vector of weight values.

The migration tool uses the “weighted” form of histogram since there is no penalty and the VCOP Kernel-C syntax always expresses histograms in weighted form.

VCOP always saturates histogram bin values as they are updated. The saturation bounds are determined by the element type of the table, as specified by the declared type of the base pointer. Thus the C7x table configuration always enables saturation, and defines the element type to correspond to the VCOP table.

The histogram operation is implemented by the `vcop_hist::whist_update()` method of the virtual machine.

5.4.8 16-Way Lookup and Histogram

VCOP memory is arranged as lines of 8 32-bit banks, for a total of 256 bits per line. This layout complements the 8-way SIMD width of table lookup operations: each SIMD lane corresponds to a lookup in one memory bank. The table layout in VCOP memory reflects this architecture.

The table layout cannot change between 8-way and 16-way SIMD. The producer of the table is independent of the consumer, so the producer need not be aware of whether the table will be read using 8-way SIMD or 16-way SIMD. So, for 16-way SIMD, only the lookup operation itself changes; the source table layout in L2 does not.

These considerations result in the following default scheme to support 16-way lookup and histogram:

- The table is assumed to be laid out in L2 memory according to the `num_tables` parameter of the `_LOOKUP` directive. This layout is dictated by VCOP’s memory architecture and `num_tables`, and is independent of the SIMD width that will be used to read the table.
- If vcop_simd=16 is specified, the lookup table operation is assumed to use twice the number of lanes as that specified by `num_tables`. For example, if `num_tables==8`, the lookup operation used 16 lanes. If `num_tables==2`, the lookup operation uses 4 lanes.
- Doubling the lookup width is enabled by duplicating the table in L1D as it is copied from L2. In effect, the N-way table becomes a 2N-way table, with the data from lanes [0,N-1] duplicated in lanes [N, 2N-1] (where N is `num_tables`).
- For a histogram operation, when the table is copied back to L2, the accumulated sums from the duplicated tables must be added together (with saturation).
- In this way the lookup throughput is doubled without requiring a change to the table layout in L2.

A few considerations apply to this scheme.

- First, the user should be aware that the table requires twice as much memory in L1 as in L2. Therefore the maximum table size that can be supported in 16-way SIMD is 16K bytes.
- Second, because of the duplication, the throughput of the copy-in/out operations is 256 bits per cycle instead of 512 bits as in 8-way mode.
- Third, like arithmetic kernels, in order to take advantage of 16-way SIMD, the kernel must be written in a SIMD-width independent way. That is, loop control (trip counts) and addressing must be in terms of `VCOP_SIMD_WIDTH`, rather than being hard-coded, or is more commonly the case, in terms of `num_tables`.

This scheme is the default, but is not beneficial for all cases of lookup operations. For this reason, `_LOOKUP` has been given an additional parameter to control this table duplication behavior, `duplication`:

```
_LOOKUP(num_tables, num_pts, duplication, table_size)
```

By default, duplication is `VCOP_SIMD_WIDTH/8`. That is, for 8-way it is 1 and for 16-way it is 2. In addition, `num_pts` may be up to `VCOP_SIMD_WIDTH`. This additional parameter is particularly useful for one table configurations. Consider the case of:

```
_LOOKUP(1, 8)
```

If the desired effect is to look up as many points as possible, this specification may be rewritten as:

```
_LOOKUP(1, VCOP_SIMD_WIDTH, 1)
```

This configuration will look up 8 points in 8 way mode and 16 points in 16 way mode from a single table in L1D. If the desired effect is to lookup sets of 8 points, the specification may instead be rewritten to:

```
_LOOKUP(1, 8, VCOP_SIMD_WIDTH/8)
```

This configuration will look up 8 points using one index from one table in 8 way mode. In 16 way mode, the table will be duplicated in memory, two index values will be used, and two sets of 8 points will be returned.

Performance

This chapter provides information about performance issues when converting from VCOP Kernel-C to C7x.

Topic **Page**

6.1 Overview

The collective design goal of the C7x ISA and the migration tool is to achieve cycle parity with VCOP on Kernel-C kernels when translated to C7x. The actual performance of a given kernel depends on a variety of factors:

- VCOP features used by the kernel
- how well the C7x ISA specifically covers those features
- the efficiency of the translation generated by the migration tool and virtual machine
- the ability of the C7x C compiler to generate efficient code from the translated code (including the virtual machine)
- outer loop code limiting the use of NLC

Of these, only the ISA itself is relatively constant.

The most significant performance issues arise from the use of LHT (lookup and histogram) operations due to the overhead of copying the table into and out of L1D, the use of OFFSET_NP1 and PDDA parallel scattering stores, and collating stores which are not well-supported on C7x.

Having said that, for many kernels the goal of cycle parity is already achieved with current tools. We have established the following general expectations:

- On average, kernels will execute on C7x with a cycles-to-cycles efficiency of about 0.7x vs VCOP.
- For kernels that can use 16-way SIMD, this doubles to about 1.4x.

6.2 Compiler Requirements

The migration tool generates C++ code that invokes methods in the virtual machine, which is also C++. As such, it relies heavily on the compiler to achieve an efficient translation. In particular, the following compiler optimizations are critical for VCOP translation:

- **Inlining.** The virtual machine is implemented as C++ classes with static inline methods. The migration tool expects the compiler to inline them fully. The `vloops()` function of a translated kernel should contain no calls.
- **Software pipelining.** This is an important key to performance on C7x. The compiler should be able to pipeline any loop generated by the migration tool for a VCOP kernel.
- **#pragma PARALLEL_LOOP.** There are no loop-carried memory dependencies in a VCOP kernel (this is a built-in property of the architecture). The migration tool uses the `PARALLEL_LOOP` pragma to convey this property to the compiler. The compiler must not limit the initiation interval of a software pipelined loop due to memory dependence.
- **NLC.** The migration tool does not directly invoke the NLC. It expresses loop control using explicit loops and relies on the compiler to collapse them using the NLC where appropriate.
- **Hoisting.** The migration tool generates many loop-invariant operations within inner loops and relies on the compiler to recognize that they are invariant and hoist them outside the loop.
- **Forward Substitution for Addressing Primitives.** The migration tool generates SE- and SA-based loads as move instructions with the SE/SA pseudo-register as the source operand. Since this operand encodes as a register operand, it can be often be propagated to where it's used. For example, the sequence:

  ```
  VMV SE0++,Vreg1
  ...
  VADDW Vreg1,Vreg2,Vdst
  ```

  can be optimized to:

  ```
  VADDW SE0++,Vreg2,Vdst
  ```

A related issue is that the migration tool sometimes generates SA-based references to objects separately from the actual indirection. The compiler should combine these. For example, the migration tool may generate:

```
int16 *p = &__sa_noadv_int(0, 4, base);     // note “address of”
...
*p = Vreg;
```

this generates:

```
ADDAW reg_base, SA0++, reg_p
...
VSTW Vreg,*reg_p
```

which should be optimized to:

```
VSTW Vreg,*reg_base[SA0++]
```

This is by no means an exhaustive list. Efficient translation relies on dozens of compiler optimizations, not all of which have been implemented.

6.3 Automatic Performance Profiling

To facilitate evaluation of the performance of translated kernels running on C7x compared to native execution on VCOP, an automatic profiling mechanism has been added to both the Kernel-C compiler (`vcc-arp32`) and the migration tool (`vcc7x`). It measures the execution cycles of both the `init()` function and the `vloops()` function on both targets so they can be compared.

The profiling mechanism relies on the built-in TSC cycle counter available on both EVE and C7x. The TSC is accessed through API calls in the RTS.

On EVE the kernel executes asynchronously. That is, the `vloops()` function only dispatches the kernel loops to VCOP but returns without waiting for them to complete. A subsequent call to `_vcop_vloop_done()` synchronizes ARP32 execution with the completion of the kernel. EVE programmers can manage this synchronization themselves by calling the `init()` function, the `vloops()` function, and the `_vcop_vloop_done()` in the client code, possibly interspersed with other operations like memory transfers.

Alternatively, programmers can use the higher-level `kernel()` function call which wraps these other calls. The `kernel()` function waits for the VCOP loop to complete before returning. Automatic profiling is supported only when the kernel is invoked through the higher-level `kernel()` API. However, users that use the lower-level calls can still use the profiling mechanisms manually by inserting calls to the timer functions themselves.

The mechanism works as follows:

- Profiling is enabled via the `--profile` switch on the VCC command line.
- VCC defines two global variables for each kernel to hold the cycle counts for the `init()` and `vloops()` functions. Given a kernel named `mykernel`, the variables are named `mykernel_init_cycles` and `mykernel_vloops_cycles`.
- In the `kernel()` function, VCC wraps calls to the `init()` and `vloops()` functions with cycle counting via `_tsc_gettime()`. In the EVE case, the vloops cycle count is taken after `_vcop_vloop_done()`, so that the elapsed time includes complete execution of the vloop command on VCOP. It stores the cycle counts in the counter variables.
- At the bottom of the `kernel()` function, VCC inserts a call to a new API function `__vcop_log_kernel_profile()` that records the accumulated cycle counts along with the kernel’s name. This function is defined in the compiler’s runtime support library.
- When the program ends (after `main()` returns) the values are automatically printed. For example:

```
kernel profiling results:
      vcop_fir_2D_short: init cycles=207 vloops cycles=64
```

If the kernel is not invoked through the high-level `kernel()` API, users may still insert cycle counting code manually and call `__vcop_log_kernel_profile()` to register the accumulated cycle counts.

On EVE, VCOP is clocked at twice the rate of ARP32. The reported cycles for EVE are ARP32 cycles, not VCOP cycles. Therefore to compare C7x cycles to VCOP cycles, the EVE cycle counts should be doubled.

6.4 Performance Options

There are a number of options available to improve performance under certain circumstances. Not all are applicable to all kernels and some may require some refactoring of source.

`--transpose` – Enables the streaming engine based transpose read transformation to generate more efficient `OFFSET_NP1` transpose sequences.

`--vcop_simd=16` – Enables 16 lanes and changes `VCOP_SIMD_WIDTH` to 16.

Warnings and Notes

A.1 Compatibility Warnings

This is a summary of the “Compatibility Warnings” throughout the document. These are cases where incorrect translation to C7x may occur.

- **Pblock contents.** Dispatch or Kernel Code that manipulates the pblock directly based on knowledge of its contents will not work. The tvals structure, which is the C7x analogue of the pblock, has different fields at different locations. (Section 2.2)
- **Pblock size.** The tvals structure for C7x tends to be larger than the pblock for VCOP. Programs which allocate the pblock without using the `kernel_pblock_size()` function may fail if the allocated space is too small. (Section 1.4)
- **Unsigned vector elements.** If a kernel relies on vector elements being treated as unsigned when bit 31 is set, the translated code may not work properly. (Section 1.4)
- **40-bit vector elements.** If a kernel depends on more than 32-bits of precision in vector elements, the translated code may not work properly. (Section 1.4)

A.2 Efficiency Warnings

This is a summary of the “Efficiency Warnings” throughout the document. These are cases where inefficient translation to C7x may occur.

- **Collate.** Stores using the collate distribution mode do not have efficient translations. (Section 5.2)
- **Parallel Stores.** Stores using OFFSET_NP1 and PDDA (`p_scatter`) distribution modes do not have efficient translations to C7x by default. (Section 5.2.2)
- **Lookup Table and Histogram Operations.** Due to the overhead of copying the table data into and out of L1D, these operations can have inefficient translations. (Section 5.4)

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265