.

T-Language specification

Specific purpose language for the THEIA GPU

Diego Valverde Garro

2/29/2012

T-Language is a special purpose language written specifically for the THEIA GPU project. The T-Language is intended to be a language that allows writing programs for the THEIA GPU in a way such that is takes advantage of the GPU features but is also easy to learn and use.

Contents

1. What is T-Language?

T-Language is a special purpose language written specifically for the THEIA GPU project. The T-Language is intended to be a language that allows writing programs for the THEIA GPU in a way such that it takes advantage of the GPU features but that it is also easy to learn and use. This is why T-Language resembles a bit of C. That being said, C/C++ programmers will hopefully feel more comfortable using this language.

2. The simplest program

Since the GPU doesn't really have such thing as a "Hello World", the simplest program would be a program that adds two numbers.

```
//This is a comment
vector MyVariable1, MyVariable2, MyVariable3;
MyVariable1 =  (1,2,3);
MyVariable2 = (4,5,6);

MyVariable3 =  MyVariable1  +  MyVariable2 ;
```

Figure 1 Function that adds two numbers

Let's look at the code from Figure 1. As usual, comments start with the "//" string, same as C++. The code declares 3 variables: MyVariable1, MyVariable2 and MyVariable3. The code then initializes MyVariable1 and MyVariable2 with some constant values.

The first thing to note is the fact that T-Language variables represent 3D vectors, in other words each variable stores a *triplet* of values. Each variable is divided into an x block, a y block and a z block. When the code from Figure 1 does the following statement:

```
MyVariable1 =  (1,2,3);
```

What is really doing is assigning the x, y and z blocks of MyVariable1 to 1, 2 and 3. This can also be written in the following way:

```
MyVariable1.x  =  1;
MyVariable1.y  =  2;
MyVariable1.z  =  3;
```

Finally the code stores the result of the addition into MyVariable3. Note that the GPU actually does 3 additions, this is MyVariable3.x = 1 + 4, MyVariable3.y = 2 + 5 and MyVariable3.z = 3 + 6.

3. Variables and data types

T-Language has two data types: vector variables or registers. The next two sections will give a brief overview of these concepts.

3.1. Vector variables

Let's start by looking at the **vector** type. The **vector** variables are 96 bit wide, and are divided into x, y and z 32 bit blocks. The next figure illustrates common uses of **vector** variables[1].

```
 vector foo;
 vector a, b, c;

a = 10;              //This is the same as a.x = 10; a.y = 10; a.z = 10;
foo.x =  a + b;     //The result from the addition is stored in foo.x
```

Figure 2 Using vector variables.

The vector variable names can be any string that is not a reserved word, please note the R[1-63] and r[1-63] are reserved words as we will see in the next section. Note also that the statement "foo.x" tells the compiler to use the 32 bit "x" block of "foo" as the destination for the addition.

[1] Note: When the user declares a **vector** variable, is basically telling the compiler: "allocate memory for this variable wherever you think is a good place". Depending on the context of the program (global variable, function frame, multiple threads enabled, etc.), the compiler can choose from several internal GPU memory locations to put the **vector** variable.

The **vector** variables can be declared and initialized in the same line. Un-initialized **vector** variables contain undefined values.

```
   vector MyVar1 =  (1,2,3);       //Defines and initializes a variable
   vector MyVar2 = (0xFF, 0b111000, 0b101010), MyVar3;       //Defines two more variables
```

Figure 3 Defining and initializing vector variables

3.2. Using the GPU registers as variables

There are times when the programmer will not want to rely on the compiler in order to allocate the variables. If this is the case, then the programmer can choose to directly use the GPU CORE registers. The GPU registers are identified using the reserved words r[0-63] or R[0-63]. Since registers are reserved words, they don't need a declaration. Example:

```
  R1 = r2 + r5*R7;
  r3.z = 0xfff;
```

Figure 4 Directly using the GPU registers

There are a couple of points to consider here. Even though the GPU registers are general purpose registers, some of these registers are used by the compiler for specific tasks. For example R0.x shall always be zero; R0.y shall always be 1, etc. The programmer can play around with the registers as long as he is aware of what the registers do.

4. Literal constants

T-Language accepts literal constants in 3 formats: decimal, hexadecimal and binary. The next snippet illustrates this:

```
//Assign a constant in decimal
   foo.x  = 1234;
//Assign a constant in HEX, the constant can be uppercase or lowercase
   bar.y  = 0xCAFE;

//Assign a constant in Binary, up to 32 bits can be used (bits assumed to be LSB)
   R1.z  = 0b111000111;

//Assign a constant representing a fractional number in fixed point scale 17
 Fixedpoint17 = 17fp3.14159265
```

```
//Assign a constant representing a fractional number in fixed point scale
 Fixedpoint8 = 8fp3.14159265

//Can even combine into a single statement
  R2 = (0b10101, 0xff, 75);
```

Figure 5 Literal constant and numeric bases

Note from the previous figure that the “0x” prefix is used to indicate a HEX constant, the “0b” prefix is used to indicate a binary constant; this is the same as in C/C++. The prefix 17fp is used to indicate that the constant represents a “scale 17 fixed point arithmetic” number. Recall from the architecture definition document [TBD] that the ALU only supports fixed point arithmetic in order to operate on fractional numbers.

5. Arithmetic expressions

Arithmetic expression consist of any combinations of +, -, *, / and sqrt().The parenthesis can be used to group sub-expressions. Example:

```
   A = (B + C*(D+2)) / (H*W – F);
   R7.y =A / sqrt( A + B *D );
```

Figure 6 Defining and initializing vector variables

It is important to note from the previous figure that the precedence of the *, and / is resolved by the compiler therefore, A + B*C is equivalent to A + (B*C).

Also note that sqrt() is not a function call, but represents an actual GPU hardware operation. However, the compiler allows to use expressions as input arguments to the sqrt (the compiler does a lot of the magic behind the scenes).

5.1. Addition and subtraction

Addition and subtraction are regular 3D vector addition/subtractions:

$$\begin{pmatrix} ax \\ ay \\ az \end{pmatrix} \pm \begin{pmatrix} bx \\ by \\ bz \end{pmatrix} = \begin{pmatrix} ax \pm bx \\ ay \pm by \\ az \pm bz \end{pmatrix}$$

```
 //Add and Sub a bunch of stuff
A = B + C + R1 – D;
```

Figure 7 Adding and subtracting

There is also an abbreviated way to add a number with the same number very similar to the C/C++ syntax:

```
  //This is the same as Foo = Foo + 5;
   Foo += 5

  //This is the same as Foo = Foo - 5;
   Foo -= 5
```

Figure 8 Adding and subtracting

5.2. Increment and decrement

T-Language features increment and decrement operators “a la C”.

Next figure illustrates this:

```
   vector V1 = (0,0,0);
  //Increment x, y and z
   V1++;               //V1 = (1,1,1)

  //Increment x only
   V1.x++;           //V1 = (2, 1, 1)

  //Decrement y only
   V1.y--;           //V1 = (2,1,0);
```

Figure 9 Increment and decrement operators

One thing to note here is the fact the T-Language only supports “pos-fix” increment and decrements (pre-fix is not supported).

5.3. Multiplication and Division

The GPU multiplication operation does 3 separate scalar multiplications on the x, y and z components of each variable. This is **not** to be confused with a cross product or a dot product. That being said, the scalar multiplication is given by the following formula:

$$A * B = \begin{pmatrix} ax & 0 & 0 \\ 0 & ay & 0 \\ 0 & 0 & az \end{pmatrix} * \begin{pmatrix} bx \\ by \\ bz \end{pmatrix} = \begin{pmatrix} ax * bx \\ ay * by \\ az * bz \end{pmatrix}$$

Division works in a similar fashion:

$$A/B = \begin{pmatrix} 1/ax & 0 & 0 \\ 0 & 1/ay & 0 \\ 0 & 0 & 1/az \end{pmatrix} * \begin{pmatrix} bx \\ by \\ bz \end{pmatrix} = \begin{pmatrix} ax/bx \\ ay/by \\ az/bz \end{pmatrix}$$

5.4. Square root

The GPU features a fast square root hardware unit. This square root unit gives an approximation of the square root value for a limited range of integer numbers represented as scale 17 fixed points.

$$\sqrt{A} = \begin{pmatrix} \sqrt{ax} \\ \sqrt{ax} \\ \sqrt{ax} \end{pmatrix}$$

Note that only the x part of the vector A is used as input for the square root. Also note that the square root output is copied in the x, y and z blocks of the result.

5.5. Sign control

The ALU allows changing the individual signs of the x, y or z components of a vector. For example:

```
vector foo,bar;
foo = (1,  1,  1) ;
bar = foo.x-yz;  //bar is (1, -1, 1)
bar = foo.-xyz;  //bar is (-1, 1, 1)
bar = foo.-x-y-z;  //bar is (-1, -1,- 1)
```

Figure 10 Defining and initializing vector variables

So in the previous example it becomes clear the each individual sign can be negated. A typical application for sign control can be a function that needs to calculate the *adjugate* of a matrix in order to find its inverse. Suppose a 3x3 matrix, the inverse can be find building a matrix of co-factors:

$$M^T = \begin{pmatrix} m11 & m12 & m13 \\ m21 & m22 & m23 \\ m31 & m32 & m33 \end{pmatrix}$$

Then the determinants are calculated:

$$M11 = \begin{vmatrix} m22 & m23 \\ m32 & m33 \end{vmatrix}, \quad M12 = \begin{vmatrix} m21 & m23 \\ m32 & m33 \end{vmatrix}, \quad etc \ldots$$

Finally the sings need to be changed, that's when sign control can become useful.

$$\begin{pmatrix} M11 & M12 & M13 \\ M21 & M22 & M23 \\ M31 & M32 & M33 \end{pmatrix} needs\ to\ apply \begin{pmatrix} + & - & + \\ - & + & - \\ + & - & + \end{pmatrix}$$

5.6. Swizzling

Operand swizzle consists of re-ordering the x, y and z blocks of the instruction input operands. Each individual x, y or z operand block can be replaced by one of the x, y or z blocks in the same operand.

Swizzling is a very powerful technique which allows the ALU to perform a variety of operations. An example of the usefulness of swizzling is matrix multiplication. Let's take for instance the following 3x3 matrix multiplication:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 1a + 2b + 3c \\ 4a + 5b + 6c \\ 7a + 8b + 9c \end{pmatrix}$$

This can be written in T-Language as follows:

```
vector Col1 = (1,4,7);
vector Col2 = (2,5,8);
vector Col2 = (3,6,9);
vector Col4 = (0xa,0xb,0xc);
vector Result;

Result = Col1 * Col4.xxx  +  Col2*Col4.yyy  +  Col3*Col4.zzz;
```

Figure 11 Using Swizzling to calculate a 3x3 matrix multiplication

Let’s do another example, this time a cross product.

$$\begin{pmatrix}1\\2\\3\end{pmatrix} X \begin{pmatrix}a\\b\\c\end{pmatrix} = \begin{pmatrix}2c-3b\\3a-1c\\1b-2a\end{pmatrix}$$

This can be written in T-Language as follows:

```
vector V1 = (1,2,3);
vector V2 = (a,b,c);
vector Result;

Result = V1.yzx * V2.zxy – V1.zxy * V2.yzx ;
```

Figure 12 Using Swizzling to calculate a cross product

As a final note, operand swizzling and sign control can be combined in the same instruction.

5.7. Scaling and Unscaling

The ALU has the ability to work with fixed point numbers or with integer numbers. For the fixed point number representation, the fixed point SCALE can be applied using the scale( ) or the unscale() operations. These operations will tell the hardware to shift the number left or right SCALE number of bits.

```
vector a1 = (1,2,3), a2 = (0xa,0xb,0xc);
vector Result;

Result = scale( unscale( a1 ) * unscale( a2 ) );
```

Figure 13 Using scale modifier on integer values

In the previous example, the vectors a1 and a2 are unscaled prior to the multiplication and then scaled back. This can be useful to avoid overflows.

The scale and unscale operations are also used to transform between fixed point and integer numbers:

```
using fixed_point_arithmetic;  //Tells the compiler to assemble for fixed point arithmetic

vector MyFixedPoint = (0x00050000, 0x00040000, 0x00000000); //this is (2.5, 2.0, 0.0) in decimal
vector MyInteger;

MyInteger = unscale( MyFixedPoint );
```

Figure 14 Transforming between integers and fixed point

One important detail about the code from Figure 14 is the line "**using** fixed_point_arithmetic;": what this does is tell the compiler to assemble so that the GPU interprets the data as fixed point arithmetic.

Fixed point arithmetic is the only way for the GPU to handle numbers with decimal since the hardware doesn't have a fixed point unit. In other words you cannot assign a vector a fractional number like this:

```
using fixed_point_arithmetic;  //Tells the compiler to assemble for fixed point arithmetic

vector MyNumber.x = 2.5;      //This is wrong!

vector MyNumber.x = 17fp2.5;  //This is OK, the compiler will transform into 2.5*2^17 = 0x50000
```

Figure 15 Using Fixed point arithmetic

6. Control flow

The Control flow specifies the order in which the operations are performed. T-Langue allows for some basic control flow as we will see next.

6.1. if-else

The syntax of the if-else statement is very similar to C/C++. Let's illustrate with some examples:

```
vector foo, bar, result;

//if block
if (foo <= bar)
{
   result = 0xaced;
}

//if and else
if (foo.xxx == 0)
{
     result = 1;
} else {
    result = 2;
}
```

Figure 16 Using if-else blocks

The first important fact to consider about T-language if statements are that when a Boolean expression is evaluated, it evaluates the anded result of the comparison on the x, y and z blocks of the arguments. In the previous example, the statement **if** (foo <= bar) is really doing if (foo.x < bar <= x) && (foo.y <= bar.y) && (foo.z <= bar.z)). If a single x, y or z block wants to be used in the the **if** statement, then the if arguments must be swizzled.

It becomes clear from Figure 16 that the use of the else part is optional. Another slight difference between T-Language and C/C++ is the that "{" and the "}" delimiters are always necessary, even if the if block encloses a single statement. This was chosen to avoid ambiguities such as this one:

```
vector a, b, n, z;

//This is not valid in T-Language
if (n > 0)
if (a > b)
     z = a;
 else
     z = b;

//This is valid in T-Language
if (n > 0) {
   if (a > b)  {
      z = a;
   } else {
      z = b;
   }
}
```

Figure 17 Mandatory use of '{' and '}' for if-else statements

6.2. Loops – while and for

The "while" and "for" keywords provide the language with basic loop functionality. Note that only the while loop is supported; the for loop is currently not supported and left for a future implementation of the compiler.

```
vector  a;
vector i = 0;

//The iteration variable is stored in i.x
while (i.x < 10)
{
      a = i;
}
```

Figure 18 if statement

The next example show the use of a **for** statement:

```
vector i, a, b;
b = 0;
//The iteration variable is stored in i.x
for (i.x = 0; i.x < 10; i.x++)
{
     a = b + i.x;
}
```

Figure 19 for statement

7. Functions and program structure

This chapter deals with function declaration and definition.

T-Language functions have return values and input arguments, and these can either be **vector** variables or register types.

An important thing to note is the fact that there is no “main” function required by the language. You can still have function called “main” for the sake of nostalgia; however code is executed from top to bottom, regardless of the location of main[2].

7.1. Basics of Functions

A function is a section of code which can be called by other code sections in order to perform a specific task. T-Language functions can take from zero up to six input parameters[3]; each parameter is passed as a copy.

Functions can return a single value. The return value can represent a 3D vector or can point to the address of a memory location.

Next is an example of the definition of a function which does not take any input arguments and does not return a value.

```
function MyFunction()
{
    vector a, b;
   if (a = b) {
```

[2] This resembles the way Perl programming language executes code.

[3] More that this slows down the Hardware, use pointers instead.

```
         return 1;
    } else {
        return 0;
    }
}
```

Figure 20 Function definition

7.2. Function return values

There are several differences between T-languages and C/C++ regarding function definitions. The first difference is that the return value type is not specified (because the size of the returned value is always that of a vector variable). The second difference is the use of the keyword **function** in order to let the compiler know that this is function definition. Also note the use of the reserved keyword **return** that is used to return values. A function can return vector variables, registers or constants.

```
//return constant
function foo()
{
        return 0xcafe;
}

//return expression
function bar()
{
        vector a,b;
        return (a+b);
}

//return register
function boo()
{
    return R7;
}

//return nothing
function weee()
{
    return;
}
```

Figure 21 return from functions

7.3. The ‘exit’ reserved word

By default, THEIA programs run once for each pixel in the scene projection plane. This means that each program will potentially run many times (and many instances of the code can be running in parallel). In order to specify when an instance of the code has to stop running, the **exit** reserved word is used.

```
function main()
{
  …

  if ( all_done == 1)
 {
    exit;
 }

…
}
```

Figure 22 The exit reserved word

Note that the **exit** command will end the current instance of the code that is running, but will not stop the subsequent instances that can potentially be triggered for other scene elements[4].

7.4. Scoping and function frames

An important fact about functions is that the variables declared within a function can only be accessed by that function. In other words functions also limit the scope of the vector variables. Global variables are not allowed.

```
vector GlobalVariable;                //No globals are allowed!

function Function1()
{
  vector a,b;                         //these are local to this function
  return (a + b );
}

function Function2()
```

[4] This also depends on the CP (control processor) code; see the control processor chapter and API programming model for more information.

```
{
  vector c,d;                                   //these are local to this function
  return ( c – d);
}
```

Figure 23 Scoping

7.5. Function parameters

Functions parameters are always pass by value (as a copy into a temporary register). Up to six parameters can be passed to functions (more than this will generate a compilation error).

```
function main()
{
  vector a = (1,2,3);
  vector b = (4,5,6);
  vector c;
 c = WeirdAddition(a,b);

}


function WeirdAddition( Argument1, Argument2 )
{
    return (Argument1.y-yy + Argument2.-zy-x );
}
```

Figure 24 Scoping

7.6. About recursion

Even if the T-language allows writing recursive functions, this is highly **discouraged**. The reason why recursion should generally be avoided is that, for performance reasons, the GPU ALU stores the vector variables in the CORE's internal register file instead of accessing the CORE external memory.

7.7. Threads

The THEIA GPU has hardware support for multithreading. In order to support launching multiple threads, the T-Language introduces two more special keywords: **thread** and **start**. The **thread** keyword is used to declare a section of code as a thread and the **start** keyword is used to start the execution of a thread as illustrated in the next figure.

```
//------------------------------------------------------
function main()
{
        //Start the second thread
        start MyThread();

        …

        i = 0;
        while (i.xxx < 10)
        {
                a = b + i;
                i++;
        }

        exit ;
}
//------------------------------------------------------
//Threads cannot take input arguments
thread MyThread()
{
        vector a = (1,2,3),b,i,expected_result = (10,11,12);
        i = 0;
        while ( i.xxx < 10)
        {
                b = a + i;
                i++;
        }
        if (b != expected_result)
         {
                 r67 = 0xdead;
         } else {
                 r67 = 0xaced;
         }
}
//------------------------------------------------------
```

Figure 25 threads

In the previous example the code in the **main** function and code in the **MyThread** function are simultaneously run by THEIA as two separate hardware threads.

8. Pointers and arrays

TBD.

9. I/O operations

As described in the THEIA architecture specification [TBD], the GPU can access external read or write locations to either read in program parameters or to store program results. In order to access external resources, the T-Language syntax introduces two special keywords: the **in** keyword and the **out** keyword.

```
#define TEXTURE_MEMORY_START_ADDRESS 0x0AC;

function main()
{
  vector Color =  (0,0,0), i;
  vector Address = TEXTURE_MEMORY_START_ADDRESS;
  for ( i.x = 0; i.x < 10; i.x++;)
  {
       Color = in[ Address ];
      out[ Address ] = Color;
  }

}
```

Figure 26 Scoping

The code from the previous figure illustrates how to read from the external memory and how to write into the external memory. The first thing to note about the previous example is the fact that there are separate keywords to read and write data as opposed to having a single IO construct for both reading and writing. This has been intentionally done this way to emphasize the fact that each THEIA Vector processor unit follows a “CROW PRAM” complaint paradigm. Recall from the architecture specification document that CROW stands from Concurrent-Read Owned- Write parallel random access machine. The CROW paradigm allows multiple GPU VP cores to concurrently read from a common shared memory while each VP also has its single and unique write-only storage. This is why there are separate **in** and **out** reserved keywords: the **in** keyword is used to specify that the code running in the current VP core will read information from the external shared memory, whereas the **out** keyword specifies that the VP core will write in its own private “Owned” Memory (which is physically a different memory from the one used to read from).

10. The preprocessor

The T-Language compiler features a simple pre-processor. The preprocessor allows for simple macro replacements as well as some modest flavor of macro functions. The next example illustrates these concepts.

```
#define MyMacroVector0 R2
#define MyMacroVector1 R3
#define MyMacroVector2 R4

#macro C = CrossProduct( A, B )
            R7 = A.yzx * B.zxy;
            R8 = A.zxy * B.yzx;
            C = R7 - R8;
#endmacro

//------------------------------------------------------------------------------------------

function main()
{
  vector Normal, tmp1, tmp2;
   tmp1 = MyMacroVector1 – MyMacroVector0;
   tmp2 = MyMacroVector2 – MyMacroVector0;

   Normal = CrossProduct( tmp1,tmp2 );

}
//------------------------------------------------------------------------------------------
```

Figure 27 Using macros

The first thing to note in the previous example is the use of the keyword **#define**. Using **#define** simply does a textual substitutions throughout the file. The second option is to use the special keyword **#macro/#endmacro**, this allows to define a simple macro substitution with parameters.

10.1. File inclusion

The preprocessor also allows including files. What the preprocessor does is to simple paste the included code where it finds the #include keyword[5].

Let's assume that the a file called "MyDefinitions.tsh" contains all of the definitions we need. The file is defined as follows:

```
…
#define CameraPostion               R47
#define GlobalIlluminationIndex    R48
#define AnsiotropicGlobalWeight R49
…
```

Figure 28 the file to be included

Now, the next code shows how to include the previous file:

```

#include "Mydefintions.tsh"

//---------------------------------------------------------------------------------------------

function main()
{
CameraPostion.x = 0;

}
//---------------------------------------------------------------------------------------------

```

Figure 29 including files

11. Appendix A - Examples

This section provides some simple examples to illustrate the use of the T-language. The code from this examples was not meant to be efficient, but instead it was meant to help the reader getting familiar with the language syntax and features.

11.1. Example 1 – Solving a system of equations

Write a T-Language program that use Cramer's rule to solve the next system of linear equations:

[5] C/C++ preprocessor actually works in a similar way.

$$\begin{gathered} x + 3y - 2z = 5 \\ 3x + 5y + 6z = 7 \\ 2x + 4y + 3z = 8 \end{gathered}$$

Solution:

Using Crammer’s rule the solution is given by the determinants:

$$x = \frac{\begin{vmatrix} 5 & 3 & -2 \\ 7 & 5 & 6 \\ 8 & 4 & 3 \end{vmatrix}}{\begin{vmatrix} 1 & 3 & -2 \\ 3 & 5 & 6 \\ 2 & 4 & 3 \end{vmatrix}}, \qquad y = \frac{\begin{vmatrix} 1 & 5 & -2 \\ 3 & 7 & 6 \\ 2 & 8 & 3 \end{vmatrix}}{\begin{vmatrix} 1 & 3 & -2 \\ 3 & 5 & 6 \\ 2 & 4 & 3 \end{vmatrix}}, \qquad z = \frac{\begin{vmatrix} 1 & 3 & 5 \\ 3 & 5 & 7 \\ 2 & 4 & 8 \end{vmatrix}}{\begin{vmatrix} 1 & 3 & -2 \\ 3 & 5 & 6 \\ 2 & 4 & 3 \end{vmatrix}}$$

Recall that the determinant of a 3x3 matrix is given by:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = aei + bfg + cdh - ceg - bdi - afh$$

A possible code that solves this problem would look like this:

```
//------------------------------------------------------------------------------------------------
//Uses Crammer’s rule to solve the system of equations
function main()
{
vector ResultX, ResultY, ResultZ;
vector A,B,C,D;

vector R1,R2,R3;
R1 = (1,3,-2);
R2 = (3,5,6);
R3 = (2,4,3);
//Calculate the divisor
D = Determinant3x3( R1, R2, R3);

R1 = (5,3,-2);
R2 = (7,5,6);
R3 = (8,4,3);
A = Determinant3x3( R1, R2, R3);

R1 = (1,5,-2);
R2 = (3,7,6);
R3 = (2,8,3);
B = Determinant3x3( R1, R2, R3);

R1 = (1,3,5);
R2 = (3,5,7);
R3 = (2,4,8);
C = Determinant3x3( R1, R2, R3);

ResultX = A / D;
ResultY = B / D;
ResultZ = C / D;

}
//-------------------------------------------------------------------------------------------
//Returns a 3x3 determinant, represent the matrix a row vectors
function Determinant3x3( Row1, Row2, Row3 )
{
vector temp1, tmp2;
//Note how multiplications can be done in parallel (it takes 1 clock cycle to calculate the 2 MULs)
temp1 = Row1 * Row2.yzx* Row1.zxy;
temp1 = temp1.xxx + temp1.yyy + temp1.zzz;

temp2 = Row1.xyz * Row2.yxz* Row2.xzy;
temp2 = temp2.xxx + temp2.yyy + temp1.zzz;

return (temp1 – temp2);

}
//-------------------------------------------------------------------------------------------
```

Figure 30 Example of using Crammer's rule to solve a simple system of equations

11.2. Example 2 – Converting a triangle into a plane

Write a T-Language code to convert a triangle (represented by three 3D points P1, P2 and P3) to an infinite plane represented by a normal vector and distance from origin.

Solution:

Build a plane from a triangle is often used to create the planes of a clipping frustum. We can use the cross-product of 2 of the vectors to obtain the orthogonal vector. To compute the distance we can use the dot product to compute the distance along the normal.

```

#macro C = DotProduct( A, B )
```

```
            R7 = A * B;
            R7.x = R7.xxx + R7.yyy;
            C = R7.xxx + R7.zzz;
#endmacro

#macro C = CrossProduct( A, B )
            R7 = A.yzx * B.zxy;
            R8 = A.zxy * B.yzx;
            C = R7 - R8;
#endmacro

//-------------------------------------------------------------------------------------------

function main()
{
  vector Normal, Distance, tmp1, tmp2;


   tmp1 = P2 – P1;
   tmp2 = P3 – P1;
   Normal = CrossProduct( tmp1,tmp2 );
   Normal = Normalize( Normal );
   Distance = DotProduct( -Normal, P1 );

}
//-------------------------------------------------------------------------------------------
function Normalize( Vector )
{
   Vector = Vector * Vector;
   return (Vector / sqrt( Vector.xxx + Vector.yyy + Vector.zzz ));
}
//-------------------------------------------------------------------------------------------
```

Figure 31 Calculating a plane from a triangle

12. T-Language for the GPU control processor

The previous sections have been dedicated to specify the high level language for the GPU vector processing units (VP). The GPU also features a special processor dedicated to load balancing and resource allocations called the control processor (CP).

The CP has a very limited set of arithmetic operations, its main purpose is to distribute code and data to the VPs and also to start and stop VP threads, therefore a special subset of the T-Language is dedicated exclusively to program the CP.

12.1. The simplest CP program

The next code snippet presents a typical CP program.

```
#include "code_block_header.thh"
#define VP_DST_CODE_MEM (1<<31)
#define VP02             2

 scalar SrcOffset = TEST_OFFSET, DstOffset;
 DstOffset = (0x0 | TEST_SIZE | VP_DST_CODE_MEM );
 copy_data_block < VP02 , DstOffset  ,SrcOffset>;


 //wait until queued block transfers are complete
 while ( block_transfer_in_progress ) {}

 start <VP02>;

 exit ;
```

Figure 32 Typical CP program

As you can see from the previous figure, the general syntax of the T-Language is kept, but there are a few differences. The first difference is that CP does not operate with vectors but with scalars. A each scalar is a 32 bit value and represents a single variable. Also the CP is not a vector processor and does not understand the concept of arrays.

The CP can perform basic arithmetic operations such as additions and subtractions, but is cannot perform divisions or multiplications. On the other hand, the most common bitwise operations such as AND, OR, NOT etc. are supported.

The most important feature of the CP is the ability to schedule "block transfers" of code or data between the MCU and the VPs and to start or stop specific VP threads.

The **copy_data_block** reserved keyword is used in order to schedule a block transfer operation and the "start" reserved word is used to start the execution of an specific a VP unit.

The code from Figure 32 performs the following operations:

Schedules the transfer of TEST_SIZE data blocks from offset TEST_OFFSET in the main memory into the code memory (VP_DST_CODE_MEM) of vector processor 2 (VP02). In other words, it is plainly loading VP02 with some code to execute. The transfer of these blocks of data can take multiple clock cycles, and VP02 cannot start executing until the necessary code has been loaded into the instruction memory, therefore the CP just have to wait until this operation is finished[6].

The **block_transfer_in_progress** internal function will return 1 as long as there are pending block transfer requests queued in the MCU therefore the sentence **while** (**block_transfer_in_progress**) {} will simply loop until the transfer is finished.

Next the **start** <VP02>; sentence will send a command to the VP02 telling it that it is time to start executing the main thread with the code we just loaded.

Finally, the **exit** ; sentence marks the end of the CP program (but the VPs can still run in the back ground).

12.2. Variables and data types

The only CP data type is the scalar. The scalar represents a single 32 bit word.

```
 scalar var1,var2,var3 = 10;
var1 = 0xcafe;
var2 = 0b1010101;
```

Figure 33 CP data types

The variable initialization works as usual, but again only scalar values, no 3D as it was the case with the VPs.

[6] But it can still do some other block transfers or run control code in the meantime.

12.3. Arithmetic expressions

A limited number of arithmetic expressions are available to the CP. For more complex expressions the VPs must be used.

```
 scalar s1,s2,s3;
s1 = 2;
s2 = 2;
s3 = s1 + s2;          //Addition
s3 = s1 – s2;         //Subtraction
s3 = s1 | s2;         //Bitwise or
s3 = s1 & s2;         //Bitwise and
s3 = ~s2;             //Bitwise two complement
```

Figure 34 CP arithmetic operations

As usual the expression can be combined and grouped using parenthesis.

```
 scalar s1,s2,s3;
s1 = s3 + s2 – (s1 & s2 << s1) | s2;
```

Figure 35 CP expressions

12.4. if-else

The if-else statements work as usual

12.5. Loops while and for

WIP

1.1. CP special keywords

WIP.

Note: also remember to talk about threads and mailboxing!