Lesson 2 - SPU Communication and DMA

Summary

This lesson will demonstrate basic SPU / PPU communication and SPU DMA, by showing how to send Square Matrices to the SPU to be multiplied together, and send the answer back to the PPU for display.

New Concepts

SPU Signalling, memory alignment, SPU DMA commands

Tutorial Overview

In this lesson, we are going to use an SPU to multiply two matrices together, and return the result. In order to do this, we need to build on the concept of Event Queues introduced in the previous lesson, and introduce the concept of SPU *signals*. Using these we can 'wake up' an SPU and tell it to process something. As SPUs can't directly access main memory, in order for our SPU to multiply our matrices together, we need to use DMA commands to copy the matrices to its local store. We must then DMA the answer back into main memory, and inform the PPU that the processing is complete, via the Event Queue system introduced in lesson 1.

Matrices

A matrix is simply a two dimensional array of numbers, with the y-axis denoted as 'columns', and the x-axis as 'rows'

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} row1 = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} column1 = \begin{bmatrix} 1 & 5 & 9 & 13 \end{bmatrix}$$

For the purposes of this tutorial, our matrices are represented simply as a a pointer to a contiguous area of memory containing the floating point values that make up our matrix. To keep things simple, we'll only be considering 'square' matrices, i.e. matrices with the same dimension for both their row and column size. Although the code outlined in this lesson will work with square matrices of any dimension, we will be using a dimension of 4, which gives the matrices a small memory footprint perfect for manipulation on the SPU. Using 4x4 matrices also serves another purpose - one you may be familiar with if you've used OpenGL. All graphical transformations such as rotation, translation, and scaling can all be kept in a single 4x4 transformation matrix. This method of storing translations is used in OpenGL, Direct3D, and the PSGL and GCM libraries used on the PlayStation, so knowing how to manipulate them will be very useful to you.

Matrix shared header

Start by making a PPU project called MatrixPPU, and an SPU project called MatrixSPU. In this tutorial, we will be using data structures to pass information between the SPU and PPU. This means that both our SPU and PPU programs are going to need to know the layout of the data structure. To do this, we are going to put the struct definition in a header file, and **#include** it in both our PPU and SPU projects. Although it could go in either project, we're going to put the header file in the MatrixPPU project. For convenience, we're also going to stick some matrix manipulation functions in this header file, too, so that both the PPU and SPU can perform the same operations on the matrices being passed around.

```
1 #define SPU_SIGNAL_PORT    2
2 #define DATA_ALIGN         16
```

matrixshared.h

The first two lines in our header file are a couple of macros - as with lesson 1, these will be explained as they are used. After our macros, we are going to define the data structure that we will be passing between the PPU and SPU. Let's take a look at it...

```
3 struct spu_data_t {
4    float *  matrixA;
5    float *  matrixB;
6    float *  matrixC;
7
8    unsigned int   dimension;
9 };
```

matrixshared.h

It should be pretty simple to understand. It has pointers to 3 arrays of floats that represent our matrices, and a value to store the matrices dimension - remember, we are using 'square' matrices, so we only need one dimension. Negative dimensions wouldn't make much sense, so we store this value as an **unsigned int**. Bear in mind that this structure does NOT contain the actual matrices, only *pointers* to them. The reasons for this will become clear when we see how the PPU and SPU communicate, and how the SPU accesses memory.

After the data structure, we are going to define three functions that manipulate our matrices. By placing them in a shared header file, they will be accessible to both the PPU and the SPU programs. First up, we have a function that will turn a given matrix into an 'identity' matrix. The result of any matrix a multiplied by an identity matrix is equal to a. An identity matrix has a value of '1' across its diagonal, and '0' elsewhere:

$$identity = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
10 void matrix_to_identity(float *m,unsigned int dimension) {
11    for(unsigned int i = 1; i < (dimension*dimension)-1; ++i) {
12       m[i] = 0.0f;
13    }
14
15    for(unsigned int i = 0; i < dimension; ++i) {
16       m[i+(i*dimension)] = 1.0f;
17    }
18 }
```

matrixshared.h

The **matrix_to_identity** function is pretty simple. It takes in a pointer to a matrix, and its dimension. The loop on line 11 sets nearly all of the matrices values to 0 - note how it skips the first and last values, as we know they are always going to be set to 1 in an identity matrix. The loop on line 15 sets each diagonal element of the matrix to 1. Note how in the matrix index we use 'i+(i*dimension)' - this simple equation skips to the next column every time i is incremented ('i+'), and also skips to the next row ('i*dimension'). It's handy way to 'walk' through a linear set of values as if they were seperate rows and columns.

Next up, we have the matrix multiplication function. For those who don't know, when multiplying two matrices together, each value of the resulting matrix is the dot product of the row of the first matrix and the column of the second.

$$\begin{bmatrix} A & B & C & D \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{bmatrix} \cdot \begin{bmatrix} . & . & W & . \\ . & . & X & . \\ . & . & Y & . \\ . & . & Z & . \end{bmatrix} = \begin{bmatrix} . & . & p & . \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{bmatrix} \quad p = \left[A.W + B.X + C.Y + D.Z\right]$$

Our function takes in a dimension value, and 3 matrices - the two we will be multiplying together, and a third, where the output matrix will be stored. We can multiply any sized square matrix using three for loops - one to walk the rows, one to walk the columns, and one to perform the dot product.

```
19 void matrix_mult(float* inA, float* inB, float* out,
20                   unsigned int dimension) {
21    for(unsigned int r = 0; r < dimension; ++r) {
22       for(unsigned int c = 0; c < dimension; ++c) {
23          out[c + (r*dimension)] = 0.0f;
24          for(unsigned int i = 0; i < dimension; ++i) {
25             out[c + (r*dimension)] +=
26             inA[(r*dimension)+i] * inB[c+(i*dimension)];
27          }
28       }
29    }
30 }
```

matrixshared.h

So for every column and every row, we set the element to 0 (line 23), and then calculate the dot product, and put the result in the relevent element of our third matrix (line 25). You should be able to see why multiplying a matrix by an identity matrix returns the original matrix.

As multiplying identity matrices together is not particularly useful, we need a function that will load our matrices up with some test data. The following function will set each element in the matrix to a value equal to that of its position in the array. Admittedly that's not really **that** useful, but it will at least show that our matrix multiplication works!

```
31 void matrix_to_test_data(float*mat,unsigned int dimension) {
32    for(unsigned int i = 0; i < dimension*dimension; ++i) {
33       mat[i] = i;
34    }
35 }
```

matrixshared.h

This should be pretty self explanatory! We walk the elelments, setting them to an incrementing value. That's everything we need in our shared header file, so now we'll take a look at what we need to put in our PPU code.

Matrix PPU Code

As with our previous lesson, we'll start by looking at the **includes** and **defines**, and then move on to the **main** function.

Includes and definitions

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <spu_printf.h>
4  #include <sys/spu_initialize.h>
5  #include <sys/spu_utility.h>
6  #include <sys/paths.h>
7  #include <sys/ppu_thread.h>
8  #include <sys/spu_thread.h>
9  #include <sys/event.h>
10
11 #include "matrixShared.h"                  //This is new!
12
13 #define MATRIX_DIMENSION      4            //This is new!
14 #define STACK_SIZE            4096
15 #define PRIO                  200
16 #define MAX_PHYSICAL_SPU      1
17 #define MAX_RAW_SPU           0
18 #define SPU_SELF  "/MatrixSPU.self"        //Note the new filename...
19 #define SPU_PROG  (SYS_APP_HOME SPU_SELF)
20
21 void ppu_thread_entry(uint64_t arg);
22 void print_matrix(float*mat, unsigned int dimension); //This is new!
23
24 SYS_PROCESS_PARAM(1001, 0x10000)
```

matrix.ppu.cpp

Not a *quite* copy and paste from lesson 1, there's *four* things different! we have an extra **#include** statement on line 11, including our shared header file so that we have access to our new data structure. The **SPU_SELF** macro on line 18 has been updated to have the name of the SPU program we're going to write. Finally, we we have a new macro definition on line 13, and a new function declaration on line 22. The macro defines a default value for our matrix dimension, and the **print_matrix** function declaration does what you'd expect it to - prints out a matrix. Why isn't this function in the shared header file with the other matrix functions? Cast your mind back to the previous lesson, the SPU doesn't have a **printf** function, it has an **spu_printf** function - so functions using printf won't compile on the SPU. Other than these changes, everything else is as it was in lesson 1, so we can move on to the **main** function itself.

Main Function

The start of our **main** function should be pretty familiar...

```
25 int main(void) {
26    sys_spu_thread_t                thread_spu;
27    sys_spu_thread_group_t          thread_group;
28    sys_event_queue_t               signal_queue;
29    sys_event_queue_attribute_t     queue_attr;
30    sys_event_t                     event;
31    int                             return_val;
32    sys_spu_image_t                 spu_img;
33    printf("PPU: Matrices!\n");
```

matrix.ppu.cpp

All that's different is we now have an **sys_event_t** variable in our main function. We events in our PPU thread in lesson 1, but we aren't using PPU threads this time around, so we'll define our event here, instead. After our variable declarations, we can initialise our SPU, create our SPU group, and load in our SPU code.

```
34    return_val = sys_spu_initialize(MAX_PHYSICAL_SPU, MAX_RAW_SPU);
35    if (return_val != CELL_OK) {
36       printf("PPU: Couldn't initialise SPUs!\n");
37       exit(return_val);
38    }
39
40    sys_spu_thread_group_attribute_t group_attr;
41    group_attr.type=SYS_SPU_THREAD_GROUP_TYPE_NORMAL;
42
43    return_val = sys_spu_thread_group_create(&thread_group,
44          MAX_PHYSICAL_SPU, 100, &group_attr);
45    if (return_val != CELL_OK) {
46       printf("PPU: SPU thread group create failed %i\n", return_val);
47       exit(return_val);
48    }
49
50    return_val = sys_spu_image_open(&spu_img, SPU_PROG);
51    if (return_val != CELL_OK) {
52       printf("PPU: sys_spu_image_open failed %i\n", return_val);
53       exit(return_val);
54    }
```

matrix.ppu.cpp

Have a feeling of Dèjá vu? We initalise our SPU, create an SPU thread group, and load our SPU code, just like in lesson 1. Now we can initialise our SPU thread, attach an Event Queue to it, and start up our SPU thread.

```
55    sys_spu_thread_attribute_t thread_attr;
56    sys_spu_thread_argument_t thread_args;
57
58    return_val = sys_spu_thread_initialize(&thread_spu, thread_group,
59                              0, &spu_img, &thread_attr,
60                               &thread_args);
61
62    if (return_val != CELL_OK) {
63       printf("PPU: SPU thread init failed: %i\n", return_val);
64       exit(return_val);
65    }
66
67    sys_event_queue_attribute_initialize(queue_attr);
68
69    return_val = sys_event_queue_create(&signal_queue, &queue_attr,
70                         SYS_EVENT_PORT_LOCAL, 127);
71    if (return_val != CELL_OK) {
72       printf("sys_event_queue_create failed %i\n", return_val);
73       exit(return_val);
74    }
75
76    //connect the queue to the SPU thread
77    return_val=sys_spu_thread_connect_event(thread_spu, signal_queue,
78                                 SYS_SPU_THREAD_EVENT_USER,
79                                 SPU_SIGNAL_PORT);      //Different!
```

```
80
81    if (return_val!=CELL_OK) {
82       printf("PPU: SPU Queue connect failed: %i\n", return_val);
83       exit(return_val);
84    }
85
86    return_val = sys_spu_thread_group_start(thread_group);
87
88    if (return_val != CELL_OK) {
89       printf("PPU: SPU thread group start failed %i\n", return_val);
90       exit(return_val);
91    }
```

matrix.ppu.cpp

So far so similar...we initialise our SPU thread, initialise our Event Queue, connect it to our SPU thread, and start our thread group, just the same as we did in lesson 1. Note that this time we are using the **SPU_SIGNAL_PORT** macro that we defined in our header file. It's set to a value of 2, whereas in lesson one we used a macro set to a value of 1. Why's this? If you remember, a port number of 1 is used by Sony's undocumented SPU printf handling functions, so we should avoid using it if we want to ever use SPU printfs in our code. We can otherwise use any port number between 0 and 63 that we like, so we keep things simple and use a value of 2.

Matrix manipulation PPU Code

Now we get on to the new stuff! In order to send data to our SPU, we need an instance of the **spu_data_t** structure we defined in our header file. We define it on line 92, but in a way you probably aren't used to. This is because of an important caveat in SPU DMA transfers - the start address of a DMA transfer equal to or over 16 bytes **MUST** be 16-byte aligned. In simple terms, that means the address pointed to must be divisible by 16, or in hex, that means the least significant bit must be 0. If we just create a local variable on the stack, we can't guarantee it begins at a 16-byte aligned address - nor can we if we use the **new** operator to create an object on the heap. So, we must use the **memalign** function - this takes in two arguments, an alignment, and a size. For the alignment, we use the **DATA_ALIGN** macro we defined in our header file, set to a value of 16. For the size, we make use of the **sizeof** operator, which we can use to determine how large our **spu_data_t** structure is. As our structure is made up of 4 variables, each 4 bytes wide, our struct has a size of 16 - just large enough to require byte-alignment! The memalign function returns the address of a chunk of byte-aligned memory of the desired size.

```
92    spu_data_t*data = (spu_data_t*)memalign(DATA_ALIGN,
93                                            sizeof(spu_data_t));
94
95    data->dimension = MATRIX_DIMENSION;
96    unsigned int data_size =
97    data->dimension *data->dimension*sizeof(float);
98
99    data->matrixA = (float*)memalign(DATA_ALIGN,data_size);
100   data->matrixB = (float*)memalign(DATA_ALIGN,data_size);
101   data->matrixC = (float*)memalign(DATA_ALIGN,data_size);
102
103   matrix_to_test_data(data->matrixA, data->dimension);
104   matrix_to_test_data(data->matrixB, data->dimension);
105   matrix_to_identity( data->matrixC, data->dimension);
```

matrix.ppu.cpp

On line 96, we set our structs **dimension** variable to **MATRIX_DIMENSION** - the macro we defined on 13 with a value of 4. on line 97, we create a local variable called **data_size** - we use this to store how much memory our matrices need. Take note at how we calculate this value, it's easy to

forget we need to use **sizeof(float)** to calculate the correct size! On lines 99, 100, and 101, we use the **memalign** function again, this time to allocate enough memory to store the three matrices we are going to send to the SPU. After that, we use the functions we defined in our shared header file to set our first two matrices to test data, and our third matrix to an identity matrix. Without doing that, our matrices would contain random data - not very useful!

After allocating and setting our matrices, it's a good idea to output them, so we can see what data they contain. We use the **print_matrix** function we delcared on line 22 to output our matrices - we'll cover the code used in this function shortly!

```
106     printf("PPU: Matrix A is:\n");
107     print_matrix(data->matrixA,data->dimension);
108     printf("PPU: Matrix B is:\n");
109     print_matrix(data->matrixB,data->dimension);
110     printf("PPU: Matrix C is:\n");
111     print_matrix(data->matrixC,data->dimension);
```

matrix.ppu.cpp

Now we get to the really fun bit, talking to our SPU and sending it some data! To show how we can wait for the SPU to process data, we're going to loop the sending and receiving of our data 10 times, as defined on line 112. on line 114, we use the SDK function **sys_spu_thread_write_snr** - this will wake up our SPU, you'll see why when we write our SPU code. Each SPU has two 'Signal Notification Registers'. These are simple 32bit registers that can be read from and written to by the PPU, but the SPU can only read - and an SPU reading one of its SNRs resets the SNR. SPUs can wait for one of their SNRs to change, and then 'wake up' to perform some data. The **sys_spu_thread_write_snr** function has 3 arguments - the first one determines which SPU will receive the signal, the second one determines which of the SPUs two SNRs the signal will be placed in, and the third one determines the value that will be placed into the chosen SNR. OK, so we can wake up an SPU, but if we can only send it 32bits, how do we send it our data structure? Hopefully you've already figured out the answer to this - we send a pointer to our data structure as the signal value! For now, let's assume that the SPU is crunching away on data and will eventually DMA the answer to our matrix multiplication into matrixC - we'll get to how it actually does this soon. Once the SPU is finished multiplying our matrices, we'll get it to send an event, which will be picked up by the Event Queue we declared on line 28 and attached to our SPU thread on line 77. **sys_event_queue_receive** is a blocking function - that means every time our loop runs, our PPU will not progress with running our code until an event is received. Handy if we don't know how long our SPU will take to process some data! On line 116 we use this function to sit and wait for our SPU to send something to our Event Queue. We then output matrixC, which should hopefully be the data the SPU has DMA'd into main memory. Once our loop finishes, on line 123 we send one final signal to our SPU, this time with a value of 0. You'll see why when we take a look at the SPU code shortly.

```
112     for(int i = 0; i < 10; ++i) {
113         printf("Sending signal to SPU\n");
114         sys_spu_thread_write_snr(thread_spu, 0, (std::uint32_t)data);
115
116         sys_event_queue_receive(signal_queue, &event, SYS_NO_TIMEOUT);
117         printf("SPU has sent an Event back!\n");
118
119         printf("PPU: Matrix C is:\n");
120         print_matrix(data->matrixC,data->dimension);
121         matrix_to_identity(data->matrixC, data->dimension);
122     }
123     sys_spu_thread_write_snr(thread_spu, 0, 0);
124
125     printf("Finished SPU loop!\n");
```

matrix.ppu.cpp

With our matrix multiplications completed, it's back to code you'll recognise lesson 1. We join and destroy our SPU thread group, destroy our Event Queue, and close our SPU code data. As we used **memalign** to allocate some memory, we must deallocate it with the **free** function. This works in the same way as the **delete** keyword you will be more familiar with, but it is considered bad form to **delete** data you have used **memalign** or **malloc** to allocated, and conversely to **free** data you have allocated using **new**.

```
126     sys_spu_thread_group_join(thread_group, 0, 0);
127     sys_spu_thread_group_destroy(thread_group);
128     sys_event_queue_destroy(signal_queue,SYS_EVENT_QUEUE_DESTROY_FORCE);
129
130     return_val = sys_spu_image_close(&spu_img);
131     if (return_val != CELL_OK) {
132        printf("PPU: sys_spu_image_close failed %i\n", return_val);
133        exit(return_val);
134     }
135
136     printf("Exiting...\n");
137
138     free(data->matrixA);
139     free(data->matrixB);
140     free(data->matrixC);
141
142     free(data);
143
144     return 0;
145 }
```

matrix.ppu.cpp

On several lines in our **main** function, we used **print_matrix** to output our matrices. Here's the code for the function. Like with the other functions that we have for working with matrices, it's based on two for loops - one for the rows and one for the columns in our matrix. For every row we use a **printf** statement to output the appropriate value. Note how we don't use the **newline** escape character, but instead the **tab** escape character - we only want to newline once every row. This outputs our matrices in a visually meaningful way, as well as keeping them reasonably neat.

print_matrix function

```
146 void print_matrix(float*mat, unsigned int dimension) {
147     for(int x = 0; x < dimension; ++x) {
148        for(int y = 0; y < dimension; ++y) {
149           printf("%f\t", (float)mat[((x*dimension) + y)]);
150        }
151        printf("\n");
152     }
153 }
```

matrix.ppu.cpp

Matrix SPU Code

That's it for our PPU-side code, but we won't get far without some SPU code to go with it. Our SPU program is going to sit in an 'infinite' loop and wait for an incoming signal from our PPU program. It will then DMA in the data structure we defined in the PPU program - remember, the incoming signal contains a pointer to our data structure. It will then use the data structure to DMA in our matrices, and then multiply them. It will then DMA out the result of this multiplication into our third matrix, and send an event to our PPUs Event Queue to let the PPU know that multiplication has completed.

Includes and definitions

As usual, we'll start with the includes and definitions. There's nothing too exciting here, but note how we include our shared header file. This path is dependant on the project name of our PPU program, which if you follow this lesson directly, should be '**MatrixPPU**'. If not, you should change the **#include** defintion appropriately. Note also the macro **DMA_TAG**. This will become important shortly...

```
1 #include <sys/spu_thread.h>    //spu_thread_exit...
2 #include <sys/spu_event.h>     //SPU events...
3 #include <stdlib.h>
4 #include <cell/dma.h>          //DMA functions...
5
6 #include "../MatrixPPU/matrixShared.h"
7
8 #define DMA_TAG 1
```

matrix.spu.cpp

Main Function

Let's start off our main function. On line 11 we define an instance of our **spu_data_t** , but in an unusual way. Remember in the PPU code we created an **spu_data_t** using the memalign function, as we couldn't guarantee how the memory was bye-aligned? Well, that's not *strictly* true. We can use the compiler attribute '**aligned**' to force a stack variable to be byte aligned. Compiler attributes have an odd looking syntax, so examine line 11 carefully. On line 13 we enter into our infinite loop, and on line 14 we read use the SDK fuction **spu_readch**. This is a blocking function, and makes the SPU sit and wait for an incoming signal on a given channel. This is the mechanism that 'wakes up' our SPU and makes it multiply some matrices that was mentioned back in the PPU code. We want to check the first SPU Signal Notification Register, so we use the SDK macro **SPU_RdSigNotify1**. **spu_readch** will place the contents of the Signal Notification Register into the variable **signal**. We than have an **if** statement, checking if **signal** is false. Why check this? Remember back on line 123 of our PPU code, when we sent a value of 0 as a signal to our SPU? That's what this if statement checks for! if signal has a value of 0, the infinite loop is broken out of, and the SPU function ends. If signal doesn't contain 0, the SPU program can assume it contains a pointer to a **spu_data_t** in main memory, so can DMA it in to its local store.

```
9  int main(void)
10 {
11    spu_data_t matrixData __attribute__ ((aligned(DATA_ALIGN)));
12
13    for(;;) {
14       uint32_t signal = spu_readch(SPU_RdSigNotify1);
15
16       if(!signal) {
17          break;
18       }
```

matrix.spu.cpp

The next 3 function calls DMA in our **spu_data_t** structure into the SPU local store.

```
19    mfc_get(&matrixData,(uint64_t)signal,sizeof(spu_data_t),
20              DMA_TAG,0,0);
21    mfc_write_tag_mask(1 << DMA_TAG);
22    mfc_read_tag_status_all();
```

matrix.spu.cpp

What on earth does all that mean? Let's take it line by line. On line 19 we use an SDK function, **mfc_get**. MFC stands for **Media Flow Controller**, the proper name for the DMA engine that allows an SPU to transfer data between its local store and main memory. The **mfc_get** function on line 19 issues a request to the SPUs MFC, asking it to DMA some data into our **matrixData** variable, from main memory at the location pointed to by our **signal** variable, of a size equal to our **spu_data_t** struct. That covers the first three arguments of the **mfc_get** function, but what about the next three? the fourth argument is a DMA 'tag', which we set to the **DMA_TAG** macro we defined on line 8 . This a simple number between 0-63 that identifies a DMA request as belonging to a specific 'group'. A DMA group is just a list of all current DMA transfers with the same tag. As you'll shortly see, we can wait for all DMA requests of a certain tag group to complete - remember, DMA requests are asynchronous, so the SPU can be processing something else while the DMA request completes! The last two variables are currently unused by the SDK - so they will always be set to 0.

Whenever we want to query the state of a DMA transfer, we must set a 'tag mask' so that the DMA query functions know which tags to query the status of. Line 21 is a call to the SDK function **mfc_write_tag_mask** , which sets the tag mask by using bit shift operators to move a '1' left the number of bits equal to **DMA_TAG**. Do you see how this works? We have 64 DMA tags (numbered 0-63), and a 64bit long tag mask, so by bit shifting a binary '1' *tag* places to the left , we create a tag mask with only the DMA tags we're interested in set to 1!

With our DMA mask set, we can tell our SPU to wait until our DMA transfer is complete. On line 22 we issue another SDK function, '**mfc_read_tag_status_all**'. This makes our SPU wait until all outstanding DMA transfers with a tag bit set in our tag mask have completed. This means that we can guarantee that our data has been DMA'd into the SPU local store before doing anything with it, as we obviously don't want to perform operations on incomplete data!

Our data has been DMA'd into local store, so now we can get on with multiplying our matrices! Or can we? Remember, our data structure only contains *pointers* to our matrix data kept in main memory. That means we're going to have to DMA them into our local store to do anything with them. the data structure does let the SPU know how large the data is, though, so we can use the **memalign** function to allocate some local store memory. You should remember the **memalign** function from the PPU code - its use is just the same on the SPU. On lines 25-27 we allocate enough memory for three matrices - the two we are going to DMA in, and one containing the answer to DMA out. We then DMA in our matrices and wait for the DMA transfer to complete, in exactly the same way that we DMA'd in our data structure on line 19.

```
23      unsigned int size =
24      (matrixData.dimension * matrixData.dimension)*sizeof(float);
25      float* matrixA = (float*)memalign(DATA_ALIGN,size);
26      float* matrixB = (float*)memalign(DATA_ALIGN,size);
27      float* matrixC = (float*)memalign(DATA_ALIGN,size);
28
29      mfc_get((void *)matrixA,(uint64_t)matrixData.matrixA,
30                size,DMA_TAG,0,0);
31      mfc_get((void *)matrixB,(uint64_t)matrixData.matrixB,
32                size,DMA_TAG,0,0);
33      mfc_write_tag_mask(1 << DMA_TAG);
34      mfc_read_tag_status_all();
```

matrix.spu.cpp

Nearly done now! We have our matrices, and can finally multiply them. on line 35, we use our **matrix_mult** function from the shared header file, multiplying our two matrices, and placing the answer in our third matrix. We must then DMA the result of our matrix multiplication back into main memory, for our PPU program to do something with. On line 39, we use the **mfc_put** SDK function. It's syntax is exactly the same as **mfc_get**, including the unused arguments! We use it to put our Local Store answer into the third matrix we have a pointer to main memory for. After waiting for the DMA to complete, we can send an event to our PPU, informing it that our SPU has finished its computation. We do this on line 44, using the SDK function **sys_spu_thread_send_event**. This function takes in three arguments - the first is a port number. We use the **SPU_SIGNAL_PORT** macro we created on line 1 of the shared header file. Our PPU Event Queue is listening on this port for incoming events - this is what the **sys_event_queue_receive** function on line 116 of our PPU code is waiting for. We can then free the memory used by our 3 matrices, just like we do in the PPU code. On line 50 is our exit function, which will only ever be called if the SPU receives a 0 for its signal - remember the **if** statement?

```
35        matrix_mult(matrixA,matrixB,matrixC,d.dimension);
36        //matC now contains the answer to matA* matB
37        //So we can DMA matC back to main memory
38
39        mfc_put((void *)matrixC,(uint64_t)matrixData.matrixC,
40                  size,DMA_TAG,0,0);
41        mfc_write_tag_mask(1 << DMA_TAG);
42        mfc_read_tag_status_all();
43
44        sys_spu_thread_send_event(SPU_SIGNAL_PORT, 0, 0);
45
46        free(matrixA);
47        free(matrixB);
48        free(matrixC);
49    }
50    sys_spu_thread_exit(0);
51 }
```

matrix.spu.cpp

That's it! All the code you need for sending data to the SPU and back. Quite a lot isn't there?

Tutorial Summary

Although you might be thinking that this lesson was a whole lot of work just to be able to multiply some matrices together, you should have learnt a lot along the way. You now know how to get the PPU and an SPU communicating with each other, as well as how to trigger the SPU into processing some data. In this case the data was matrices, but it could be anything- such as an animation skeleton that needs transforming, or pathfinding data. You've also learnt how to DMA data in and out of the SPU, including some important caveats. Next lesson we will build upon these concepts, and get multiple SPUs communicating with each other and processing data.

Further Work

1) Our struct has a size of 16 - so must be 16-byte aligned in memory. Can we make our struct smaller? What about if we change the dimension variable to a short? How big is the struct then? Why is this?

2) It'd be pretty nice to be able to use **spu_printf** to print out our matrices from the SPU wouldn't it? How many Event Queues would we need? what ports would be needed?

3) SPUs have no direct access to main memory. Do PPUs have access to the SPU local store?

How else could the SPU receive our matrix data?

3) All the code in this lesson will work with square matrices of an arbitrary size, set with the **MATRIX_DIMENSION** macro. What is the upper limit on the size of the matrices the SPU can process? Why is this?