REVISED NOTES –
PLEASE USE INSTEAD OF
ORIGINAL HANDOUT

Sieve Partitioning System for Cell Linux

A Practical Introduction

Alastair F. Donaldson

Codeplay Software Ltd.

ally@codeplay.com

Laboratory session

International Summer School on Advances in Programming Languages, Heriot-Watt University, Edinburgh

27th August 2009

Introduction

In this lab session you will use the Codeplay Sieve Partitioning System to offload parts of C++ applications to run on the SPE (Synergistic Processor Element) cores of the Cell Broadband Engine processor.

Since we do not have a lab-full of Cell BE processors, you will use the IBM Full System Simulator to execute compiled code. Unfortunately this means that you will not be able to assess the performance improvements gained by code offloading, since the *fast* mode of the simulator is functionally accurate but not cycle accurate. (Cycle accurate mode is *extremely* slow, taking around 10 minutes to simulate a simple “Hello, world!” program!)

The main focus of the lab will be on understanding the language extensions proposed by Codeplay, and how they can be used to incrementally offload and optimize code.

Bug reporting

If you come across any bugs in the system during the lab then *please* log them: make a new folder, copy all relevant files for the bug into this folder, and record the compiler command you used to reproduce the bug. Then talk to Alastair Donaldson after (or during) the session, or email him an archive of the bug folder (ally@codeplay.com) – this kind of feedback is extremely valuable.

Additionally, any suggestions for improvements to the system (related to functionality or usability) are very welcome!

Overview

The lab session is structured as follows:

Setup [estimated time: 10 minutes]

Before starting the lab proper, you will download and install the Sieve Partitioning System, and check that your machine is appropriately set up by running a simple program using the Cell Simulator.

Part 1 – Offloading image filters [estimated time: 50 minutes]

This is the bulk of the lab session, and will: illustrate the process of offloading code using sievethread blocks, demonstrate the way function pointers and multiple compilation units are handled by the language extensions, and show how offloaded code can be incrementally optimized for a particular target.

Part 2 – Exploring overloading and type inference [estimated time: 20 minutes]

Don't worry if you run out of time during Part 1. If you complete Part 1 in good time then Part 2 will illustrate the way outer pointers can be used for function overloading, and will also show how a limited form of type inference is used to help automate method duplication.

Part 3 – Experiment further! [any time remaining]

If you complete the lab work particularly fast you may wish to spend some time experimenting further with the Sieve Partitioning System, either based on experience from the rest of the lab session, or by looking at further features described in the documentation included with the lab session materials.

Acknowledgements

A massive thank you to Iain McCrone for undertaking the mammoth effort of getting the various components required for this lab session working and installed on the lab machines. Thanks to Philipp Ruemmer for technical advice, Athanasios Konstantinidis for an introduction to the Cell Simulator, and Mustafa Aswad for giving the session a dry run before the Summer School. Finally, thanks to the Codeplay Sieve team (Pete Cooper, Uwe Dolinsky, Andrew Richards, Colin Riley and George Russell) for making the Sieve Partitioning System available for use, and advising on various issues which arose during the preparation of these notes.

Setup

IMPORTANT: *please work in your home directory unless the instructions state otherwise. The examples supplied with the lab materials assume this.*

Logging into `lxpara1`

The software required for the lab is on a particular machine, `lxpara1`. Use your guest username and password to log into this machine via SSH, using the `-X` option:

```
-bash-3.2$ ssh -X guestXX@lxpara1.macs.hw.ac.uk
guestXX@lxpara1.macs.hw.ac.uk's password:
[type password]
```

Getting materials for the lab session

Download an archive of materials from the session:

```
-bash-3.2$ wget http://allydonaldson.co.uk/SummerSchoolLabSession.tar.gz
```

Decompress:

```
-bash-3.2$ gunzip SummerSchoolLabSession.tar.gz
```

Extract:

```
-bash-3.2$ tar -xvf SummerSchoolLabSession.tar
```

Installing the Sieve Partitioning System

Use *wine* (Windows emulator) to run the Sieve Partitioning System installer executable:

```
-bash-3.2$ wine SummerSchoolLabSession/SievePartitioningSystemCellLinuxv1.4.4.alpha\
@20090429.exe
```

You will be taken through a sequence of installation screens – see next page for a diagram and instructions for how to navigate through the installer.

To configure the installation to suit the Heriot-Watt lab settings, run a small batch file:

```
-bash-3.2$ source SummerSchoolLabSession/setupenvironment
```

NOTE: *Before you leave the lab, even if you leave early, please uninstall the Sieve Partitioning System by running:*

```
-bash-3.2$ wine uninstall
```

and then clicking `Uninstall`.

Compiling "Hello, world!" using the Sieve Partitioning System and GNU tools

Before using the Sieve Partitioning System for code offloading, we shall compile and execute a simple program to check that the Sieve Partitioning System, Cell SDK and Cell Simulator are working properly.

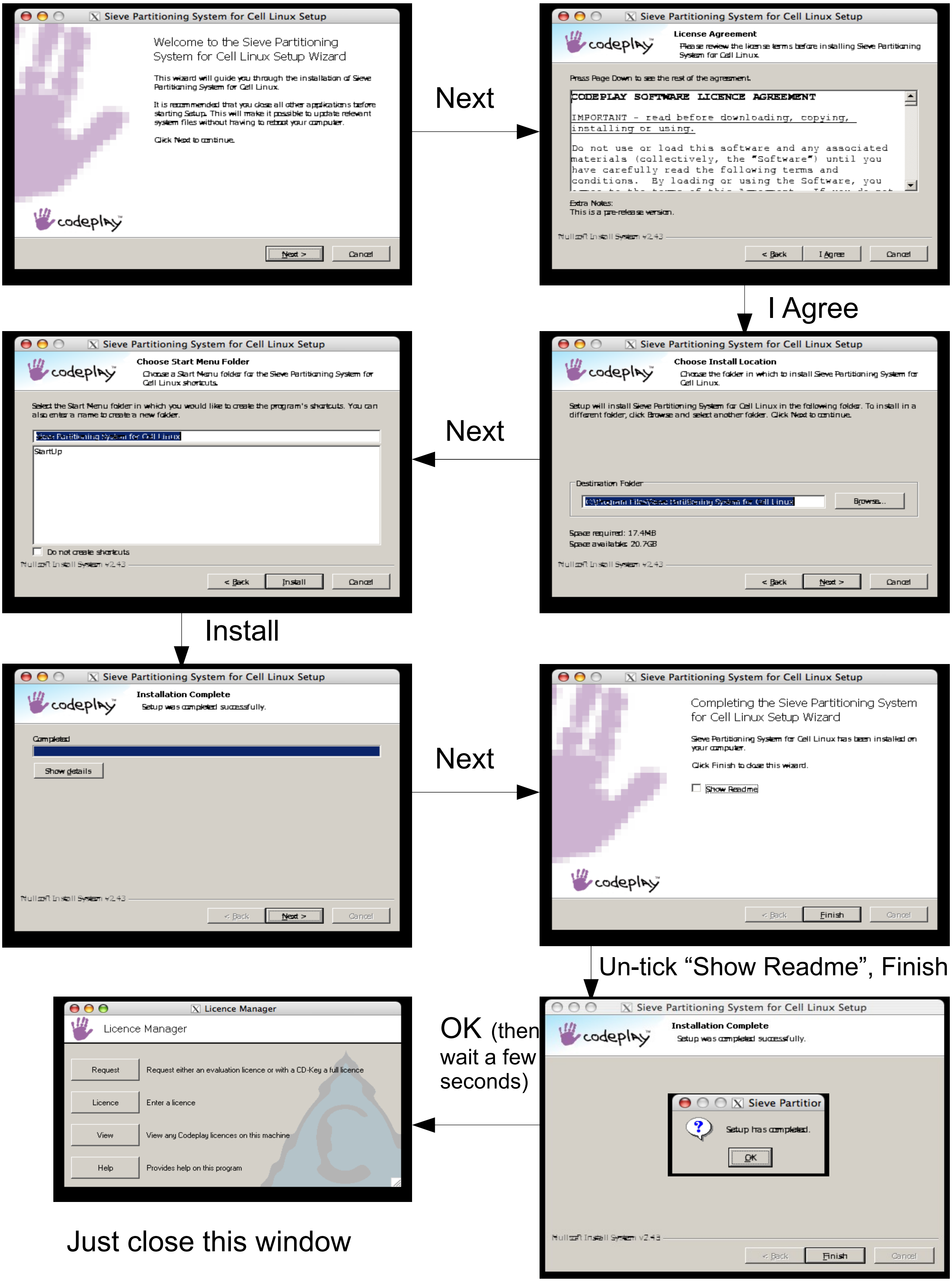
Make a file called `hello.cpp` and use your favourite editor (e.g. *emacs*) to enter a simple *"Hello, world!"* program:

```
-bash-3.2$ touch hello.cpp
-bash-3.2$ emacs hello.cpp
```

```
#include <stdio.h>

int main()
{
  printf("Hello, world!\n");
  return 0;
}
```

Use *CellSieveCPP*, the Sieve Partitioning System compiler, to compile this file:



Installation walkthrough for Sieve Partitioning System

```
-bash-3.2$ wine CellSieveCPP hello.cpp
CellSieveCPP (1.4.4 CELL on 20090429) Copyright (C) 2002-2009 Codeplay Technology Limited
and Codeplay Software Limited. All rights reserved.

hello.cpp             (CellSieveCPP) - 0 errors,     0 seconds, 9968 lines
```

TROUBLESHOOTING: Did you get this error?

```
Your licence for CellSieveCPP is not valid.
You will need to obtain a valid licence with the 'Licence Manager' program.
Go to:
http://www.codeplay.com/licence.shtml?co=44390C1044E2FB39443B003944391344391339
to get an evaluation licence. If you have a CD Key, type in:
  vectorc86 /licencecode cd-key
```

Then you probably didn't do

```
-bash-3.2$ source SummerSchoolLabSession/setupenvironment
```

above. Do this and try again.

This will create a folder called `outputc` which contains machine-generated ANSI-C files corresponding to the source program. These are compile for Cell using the *ppu-gcc* and *spu-gcc* compilers, which are part of the IBM Cell SDK. *CellSieveCPP* generates a makefile to automate this process.

Navigate to the `outputc` folder, and type `make`:

```
-bash-3.2$ cd outputc/
-bash-3.2$ make
ppu32-gcc -std=c99  -W -Wall -Wuninitialized -Wcomment -Wchar-subscripts -Wdeprecated-
declarations -Wendif-labels -Wformat-extra-args -Wimplicit -Wimport -Winline -Wmissing-
...
-DBIG_ENDIAN_MACHINE -D __USE_SWCACHE *.o -lspe2 -lsieve -lm -o main
```

This should succeed without errors, and result in a file called `main`. Although this file has execute permissions, you can't execute it as it is a binary for the Cell BE processor and will not work on your x86 machine:

```
-bash-3.2$ ./main
-bash: ./main: cannot execute binary file
```

Executing “Hello, world!” using the Cell Simulator

Open a new terminal window, and once again SSH into `lxpara1`:

```
-bash-3.2$ ssh -X guestXX@lxpara1.macs.hw.ac.uk
guestXX@lxpara1.macs.hw.ac.uk's password:
[type password]
```

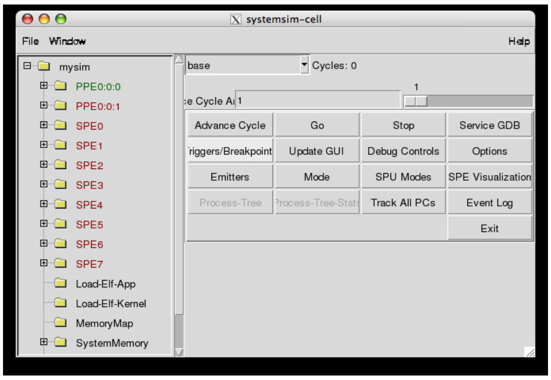
then do:

```
-bash-3.2$ source SummerSchoolLabSession/setupsimulator
```

In this new terminal window, launch the IBM Full System Simulator in GUI mode by typing `systemsim -g`:

```
-bash-3.2$ systemsim -g
=============================================================================
THIS CELL BROADBAND ENGINE SYSTEM SIMULATOR, TOGETHER WITH ALL PERFORMANCE
DATA RESULTING THEREFROM, IS PROVIDED BY IBM AND RECEIVED BY YOU ON AN "AS-IS"
BASIS, WITHOUT WARRANTY OF ANY KIND. SEE THE APPLICABLE LICENSE FOR ADDITIONAL
TERMS AND CONDITIONS.
=============================================================================
GUI Enabled
Licensed Materials - Property of IBM.
(C) Copyright IBM Corporation 2001, 2009
All Rights Reserved.
Using initial run script /opt/ibm/systemsim-cell/lib/cell/systemsim.tcl
```

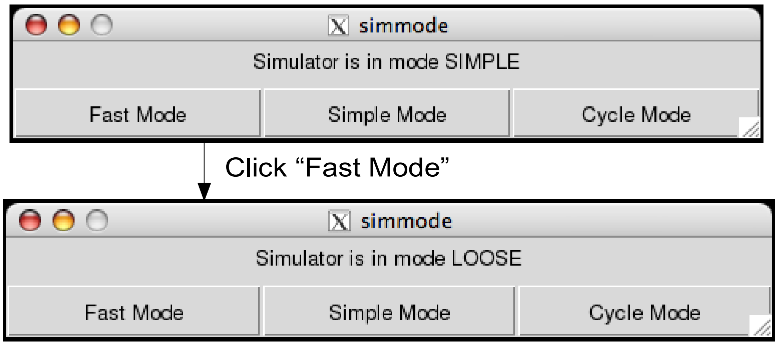
After some seconds' delay, this will open two new windows, entitled `systemsim-cell` and `mysim`:



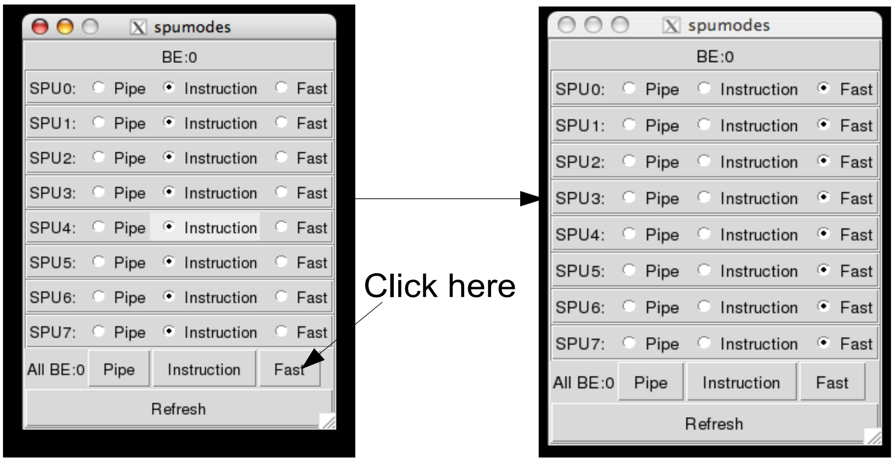
The terminal from which you launched the simulator should show this:

```
…
LOAD : ELF startup: PC=0x0000000001000000, msr=0x1000000000000000
LOAD :              gpr[1]=0x000000000FFFFF90, gpr[2]=0x0000000000000000
systemsim %
```

Click `MODE` in the `systemsim-cell` GUI to produce a `simmode` dialogue box. Click on `Fast Mode` – the simulator mode should change from `SIMPLE` to `LOOSE`:



Close the `simmode` dialogue box. Now click `SPU Modes` in the `systemsim-cell` GUI to produce an `spumodes` dialogue box. Click the `Fast` box in the bottom right hand corner, to change the mode of each simulated SPU to `Fast`:



Close the `spumodes` dialogue box.

Now click `GO` in the `systemsim-cell` GUI. This will cause the `mysim` window to boot the Fedora 9 operating system on the simulated Cell processor, ending with this output:

```
                Welcome to Fedora Fedora release 9 (Sulphur)
                Press 'I' to enter interactive startup.
eth0: bogus network driver initialization
No IRQ retreived
Starting login process
[root@(none) ~]#
```

This may take a couple of minutes. Meanwhile, the terminal window from which you started the simulator will proceed to spout garbage along the lines of:

```
   while execu..."
error in background error handler:
out of stack space (infinite loop?)
    while executing
"::tcl::Bgerror {out of stack space (infinite loop?)} {-code 1 -level 0 -errorcode NONE -errorinfo
{out of stack space (infinite loop?)
    while execu..."
35263820000000: [0:0:0]: (PC:0xC00000000005C838) :      10.8 Mega-Inst/Sec :     550.3 Giga-Cycles/Sec
35263820000000: [0:0:1]: (PC:0xC00000000005C838) :       9.4 Mega-Inst/Sec
```

Do not worry about this – this window will continue to display such output in an infinite loop. Minimize the window and ignore it for the rest of the session.

To get a binary to run in the simulator, you use a program called *callthru* to copy the binary to the simulated workspace. In the simulator terminal window (`mysim`), do:

```
[root@(none) ~]# callthru source /u1/others/guestXX/outputc/main > main
[root@(none) ~]# chmod +x main
[root@(none) ~]# ./main
Hello, world!
```

where *XX* is your guest login number.

So far so good – you have compiled a simple program for the Cell BE processor using the Sieve Partitioning System, and executed it in the simulator!

Obviously this example is trivial – it does not use the Cell SPE processors, or any of the Sieve Thread language extensions. This will be the subject of the rest of the lab.

IMPORTANT: *Do not close the simulator – you will need it for the remainder of this session.*

Part 1 – Offloading image filters

We are going to explore the Sieve language extensions by taking two simple image processing filters, and offloading them to run across all the cores of the Cell processor.

Checking out Lena

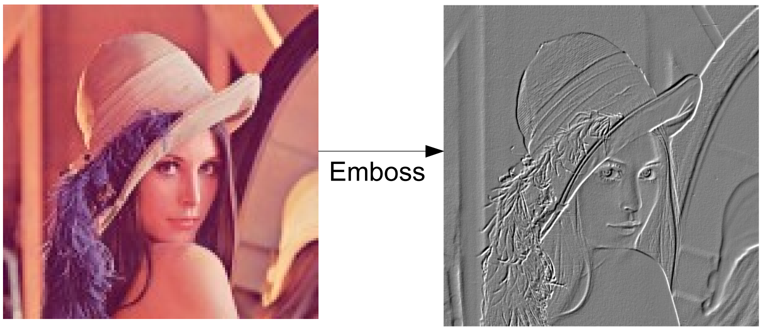
We shall apply our image processing filters to the industry-standard cropped image of Swedish model Lena Söderberg. A 128x128 pixel image of Lena is provided: `SummerSchoolLabSession/lena.ppm` (we use a small image because the Cell simulator is rather slow, even in FAST mode). Open this image using *GIMP*, and leave *GIMP* open:

```
-bash-3.2$ gimp SummerSchoolLabSession/lena.ppm &
```

GIMP takes a while to start, and you may need to go through some installation screens to get the tool to start for the first time. We shall use *GIMP* throughout the lab session to view image results, which is why it's best to leave it open.

Emboss filter on PPE only

The first image processing filter we shall consider performs embossing, producing the following effect:



Copy the following files into your home directory:

```
SummerSchoolLabSession/image/image.h
SummerSchoolLabSession/image/image.cpp
SummerSchoolLabSession/emboss_ppe/emboss.cpp
```

Have a quick look at `emboss.cpp` – it is a simple piece of C++ code which takes an input image, and produces an output image by applying a kernel to each input pixel. The details are not important for us, except that it is clear that the order in which pixels are processed is irrelevant and thus the filter is a good candidate for parallelisation.

IMPORTANT: Use `rm -rf outputc` to remove the existing `outputc` directory. *You must do this every time you recompile, otherwise you will get strange link errors.*

Compile `emboss.cpp` and `image.cpp` with *CellSieveCPP*:

```
-bash-3.2$ wine CellSieveCPP emboss.cpp image.cpp
```

(Ignore the warnings the compiler generates when it processes the Linux header files.)

Select the `mysim` window, use the *callthru* utility to copy both `main` and `lena.ppm` into the simulated workspace, and run `main`:

```
[root@(none) ~]# callthru source /u1/others/guestXX/outputc/main > main
[root@(none) ~]# callthru source /u1/others/guestXX/SummerSchoolLabSession/lena.ppm > lena.ppm
[root@(none) ~]# ./main
Usage: ./main input-ppm-file output-ppm-file
[root@(none) ~]# ./main lena.ppm output.ppm

Running emboss filter on input file lena.ppm
Writing results to output file output.ppm
```

(Note that you don't need to add execute permissions to `main` this time, since you are overwriting an existing executable file called `main`.)

To view the output image, we use *callthru* again to copy `output.ppm` back to `/u1/others/guest`*XX*:

```
[root@(none) ~]# callthru sink /u1/others/guestXX/output.ppm < output.ppm
```

Now you should be able to open `/u1/others/guest`*XX*`/output.ppm` using *GIMP*, and you should see the the embossed version of Lena as shown above.

Offloading emboss filter to a single SPE

Now we want to get the emboss filter running on the Cell SPEs. For starters, let's offload the *whole* filter to run on a *single* SPE.

Open `emboss.cpp`, and look at the body of the `emboss` function.

Enclose the whole of the body in a sievethread block:

```
void emboss( … )
{
    int handle = sievethread
    {
        const int start = ...

    };
}
```

NOTE: The closing brace of your sievethread block must be followed by a semicolon, since the block is actually a single statement from the point of view of its enclosing scope.

A sievethread block executes asynchronously as an SPE thread, which is why the block returns a handle. Before the end of the function, but after the end of the sievethread block, add a call to `sieveThreadJoin` to wait for the sievethread to complete:

```
sieveThreadJoin(handle);
```

While sievethread is a new keyword, `sieveThreadJoin` is a library function, and requires a header file, `libsieve`, to be included in `emboss.cpp`. Add this line to the top of the file:

```
#include <libsieve>
```

Now compile `emboss.cpp` and `image.cpp` with *CellSieveCPP*.

This should lead to the following error:

```
*** ERROR: (Sieve4103) Cannot access outer stack variable 'output' from inside
sievethread block without a parameter.
```

The problem is that the sievethread block refers to `input` and `output`. These are parameters to `emboss`, therefore they are located on the stack. Because sievethread blocks run asynchronously it is possible, in general (and often usual), for the calling function to return before the sievethread completes, or for the calling function to modify variables on which the sievethread depends.

To work around this, we list the variables which the sievethread requires from the enclosing scope as sievethread parameters:

```
int handle = sievethread (intput, output) …
```

This causes a copy of these variables to be passed to the sievethread on creation, avoiding problems with the original variables changing or going out of scope.

As evidence that this version of the filter really does launch an SPE thread, add:

```
printf("SPE thread started\n");
```

and

```
printf("SPE thread finished\n");
```

to the start and end of the sievethread block, and

```
printf("PPE thread waiting for SPE thread\n");
```

just before `sieveThreadJoin`.

(If you're having trouble, look at `SummerSchoolLabSession/emboss_basic_offload` for a model solution.)

Now do the “compile-run” process: compile the application with *CellSieveCPP*, run *make*, and in the `mysim` window use *callthru* to copy `main` from `/u1/others/guest`*`XX`*`/outputc`.

Run the application to produce the following output:

```
Running emboss filter on input file lena.ppm
PPE thread waiting for SPE thread
SPE thread started
SPE thread finished
Writing results to output file output.ppm
```

(Remember that since we are running the simulator in FAST mode, the time taken for execution is, unfortunately meaningless.)

Finally, use:

```
[root@(none) ~]# callthru sink /u1/others/guestXX/output.ppm < output.ppm
```

to copy the output image back, and check that it looks correct in *GIMP*.

We have shown that, with very few changes, the Sieve Partitioning System can get code to run on a Cell SPE. We didn't need to change the core code for the emboss filter *at all*. The system has automatically generated code to move data from main memory to and from the SPE local store.

Also notice that we didn't have to declare any `__outer` pointers – even though `input` is an `__outer` pointer (since it is declared outside the sievethread block) the compiler uses method duplication to automatically compile a version of `compute_pixel` which accepts an `__outer` pointer argument.

Parallelising emboss filter over 2 SPEs

Offloading to a single SPE can be a goal in itself: this frees the PPE processor so that other PPE threads can do useful work. However, for a parallelisable application such as our image filter, we can offload to multiple SPEs in parallel.

Let's start with two SPEs.

- Split the "**for(int y=start; y<end; y++)**" loop in half, and put each half in its own sievethread block (changing `start` and `end` appropriately).
- Use two handles, `handle1` and `handle2`, and have the PPE wait on each of these handles in turn at the end of the emboss function.
- Make the first sievethread block print "`SPE thread 1 started`" and "`SPE thread 1 finished`", and do similarly for the second sievethread block.
- Make the PPE print "`PPE thread waiting for SPE thread 1`" and "`PPE thread waiting for SPE thread 2`" before the respective calls to `sieveThreadJoin`.

If you have trouble, look at the model solution in `SummerSchoolLabSession/emboss_pair_offload`.

Compile and run this version of the application, to produce output like the following (the precise order of output may vary depending on the order in which the concurrent threads execute their print statements):

```
Running emboss filter on input file lena.ppm
PPE thread waiting for SPE thread 1
SPE thread 1 started
SPE thread 2 started
SPE thread 1 finished
SPE thread 2 finished
PPE thread waiting for SPE thread 2
Writing results to output file output.ppm
```

Check that the output image looks right (you may not always want to bother with this check).

Parallelising emboss filter over 8 SPEs + PPE

A major feature of the Sieve Partitioning System is that separate source code for offloaded threads and host threads is *not* required. To illustrate this, let us offload the emboss filter to run across all 8 SPEs *and* the PPE.

Let's record the fact that we have 8 SPEs, and say that we want the PPE to process 20 or so rows of our image:

```
#define NUM_SPES 8
#define PPE_PORTION 20
```

Now, in the body of `emboss`, we shall use a loop to launch `NUM_SPES` sievethread blocks, storing the associated handles in an array (note that `i` is added to the list of sievethread parameters):

```
int handles[NUM_SPES];

for(int i=0; i<NUM_SPES; i++)
{
    handles[i] = sievethread(input, output, i)
    {
        const int start = ...;
        const int end = ...;

        printf("SPE thread %d started\n", i);
        for(int y=start; y<end; y++)
```

```
        {
            /* As before */
        }
        printf("SPE thread %d finished\n", i);
    };
    printf("Launched SPE thread %d\n", i);
}
```

Figuring out the correct values for `start` and `end` is a little tricky – have a go! The above code also includes PPE messages to indicate that it has launched each SPE thread.

Now, after this sievethread creation loop, let's get the PPE to do some work:

```
printf("PPE about to process part of image\n");

const int start = ...;
const int end = ...;

for(int y=start; y<end; y++)
{
    /* As before */
}

printf("PPE processing complete\n");
```

Again, try to figure out the correct values for `start` and `end` (you'll find that due to rounding issues the PPE will process a few more than `PPE_PORTION` rows).

Then use a loop so that the PPE waits for each SPE thread to complete:

```
for(int i=0; i<NUM_SPES; i++)
{
    printf("PPE thread waiting for SPE thread %d\n", i);
    sieveThreadJoin(handles[i]);
}
```

Compile and run this example – you should see output along these lines:

```
Running emboss filter on input file lena.ppm
Launched SPE thread 0
Launched SPE thread 1
Launched SPE thread 2
Launched SPE thread 3
Launched SPE thread 4
Launched SPE thread 5
Launched SPE thread 6
Launched SPE thread 7
PPE about to process part of image
SPE thread 0 started
SPE thread 4 started
SPE thread 3 started
SPE thread 5 started
SPE thread 1 started
SPE thread 6 started
SPE thread 2 started
SPE thread 7 started
SPE thread 0 finished
PPE processing complete
PPE thread waiting for SPE thread 0
PPE thread waiting for SPE thread 1
SPE thread 4 finished
SPE thread 5 finished
SPE thread 1 finished
SPE thread 3 finished
SPE thread 6 finished
SPE thread 2 finished
PPE thread waiting for SPE thread 2
SPE thread 7 finished
PPE thread waiting for SPE thread 3
PPE thread waiting for SPE thread 4
PPE thread waiting for SPE thread 5
PPE thread waiting for SPE thread 6
```

```
PPE thread waiting for SPE thread 7
Writing results to output file output.ppm
```

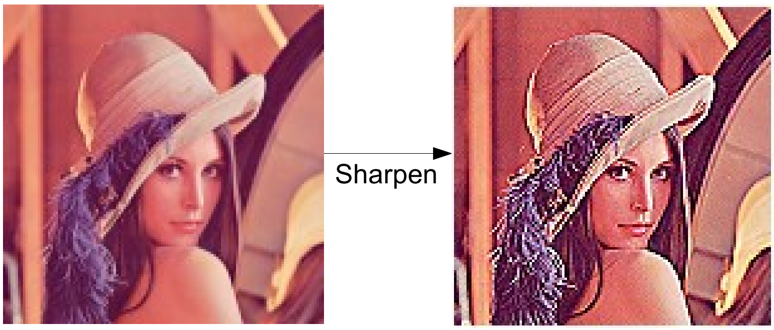
(Notice the non-deterministic order in which the threads start and finish.)

If you have got `start` and `end` wrong then the output image won't look correct – it will probably have chunks of white in it. If you have trouble then look at the model solution in `SummerSchoolLabSession/emboss_full_offload`.

The attractive feature here is that the functionality for the emboss filter did not have to be duplicated to get the filter to run on both types of processor. We have introduced a bit of duplication as the loop which applies the kernel to each pixel appears separately inside and outside the sievethread block. This can be avoided by making a function which takes start and end as parameters and encapsulates the loop – feel free to have a go at this.

Adding a sharpen filter

We now consider a larger application with an additional filter, which performs image sharpening (a more subtle effect than embossing):



Delete `emboss.cpp`, and copy `SummerSchoolLabSession/emboss_or_sharpen/emboss_or_sharpen.cpp` to your home directory:

```
-bash-3.2$ rm emboss.cpp
-bash-3.2$ cp SummerSchoolLabSession/emboss_or_sharpen/emboss_or_sharpen.cpp .
```

Have a look at `emboss_or_sharpen.cpp`. There are two effects – embossing, implemented as `emboss_pixel`, and sharpening, implemented as `sharpen_pixel`. Function `apply_effect` takes a numeric parameter specifying which effect to apply (0 for embossing, 1 for sharpening), and uses the SPEs and PPEs to apply the effect.

The `apply_effect_to_image_slice` function uses the conditional operator to decide whether to call `emboss_pixel` or `sharpen_pixel` based on the value of the `effect` parameter.

If you have plenty of time left, compile and run this example as is: compile `emboss_or_sharpen.cpp` and `image.cpp` together using *CellSieveCPP*, then do the familiar make/simulate process. Use the simulator to produce both sharpened and embossed Lena images, and use *callthru* to copy these back to be verified visually in *GIMP*.

Using a function pointer to select between filters at run-time

Although using a conditional to select between methods works OK when there are just two filters, this approach would not scale. It would be more desirable to use an array of function pointers.

Let's adapt the example to use such function pointers. First, near the top of `emboss_or_sharpen.cpp`, declare a type for pointers to functions like `emboss_pixel` and `sharpen_pixel`:

```
typedef rgb (*pixel_effect_t) (const rgb*, int, int);
```

Then modify `apply_effect` and `apply_effect_to_image_slice`, so that that parameter `effect` has type `pixel_effect_t`.

In `apply_effect_to_image_slice`, change the line which applies the effect function from:

```
output[y*WIDTH+x] = effect == EMBOSS ? emboss_pixel(input, y, x) : sharpen_pixel(input, y, x);
```

to use the function pointer parameter:

```
output[y*WIDTH+x] = effect(input, y, x);
```

Just before `main`, declare an array of effect functions, populated appropriately:

```
pixel_effect_t pixel_effects[2] = { emboss_pixel, sharpen_pixel };
```

In the body of main, change:

```
apply_effect(inputPixels, outputPixels, effect);
```

to:

```
apply_effect(inputPixels, outputPixels, pixel_effects[effect] );
```

If you get stuck, look at `SummerSchoolLabSession/emboss_or_sharpen_fp`.

Now compile and make the application – this should work. But when you try to run the application in the simulator you should get errors:

```
Running emboss filter on input file lena.ppm
Launched SPE thread 0
Launched SPE thread 1
Launched SPE thread 2
Launched SPE thread 3
Launched SPE thread 4
Launched SPE thread 5
Launched SPE thread 6
Launched SPE thread 7
PPE about to process part of image
Sieve SPU RT Error: 0, 0x10001cd8
SPE STOP AND SIGNAL
Sieve SPU RT Error: 0, 0x10001cd8
SPE STOP AND SIGNAL
Sieve SPU RT Error: 0, 0x10001cd8
Sieve SPU RT Error: 0, 0x10001cd8
SPE STOP AND SIGNAL
Sieve SPU RT Error: 0, 0x10001cd8
```
etc.

(If you do not see these errors then you probably did not delete the `outputc` folder since you compiled the previous version – do so and try again.)

The problem is that because the calls to `emboss_pixel` and `sharpen_pixel` happen indirectly via a function pointer, the compiler does not know that it needs to duplicate these methods to run on the SPEs. At runtime, the SPE tries to resolve the call, and finds that it has no matching function.

We remedy this by equipping the sievethread block with a *domain*, listing the functions we require by name.

As a first approximation, change the declaration of the sievethread block to include a domain as follows:

```
sievethread [ emboss_pixel, sharpen_pixel ] ( input, output, i, effect ) { … }
```

This tells the compiler: *"any call via a function pointer must call either `emboss_pixel` or `sharpen_pixel`, so compile them for the SPEs".*

However, this is not quite enough. The trouble is that the `emboss_pixel` and `sharpen_pixel` functions have not been declared with an `__outer` pointer for parameter `input`. So far, it has been possible for the compiler to deduce this automatically. In this scenario, we need to tell the compiler to duplicate versions of these methods which take an outer pointer.

To do this, we declare another function pointer type:

```
typedef rgb (*pixel_effect_outer_t) ( const __outer rgb*, int, int );
```

and cast the members of the domain to this type:

```
sievethread [ (pixel_effect_outer_t)emboss_pixel, (pixel_effect_outer_t)sharpen_pixel ]
( input, output, i, effect ) { … }
```

Try this – you should find that the example compiles and executes as usual. Have a look at `SummerSchoolLabSession/emboss_or_sharpen_fp_with_domains` if you get stuck.

Note that this is the first time we have had to use the `__outer` keyword!

Separating filters into multiple compilation units

In our examples so far, all the code called from within a sievethread block (apart from `printf`) has been located in one source file. This makes compilation of the sievethread block straightforward, but this approach is not feasible for large applications.

With a little more user annotation it is possible for the extent of a sievethread block to span multiple compilation units.

We will illustrate this by considering a version of our image filter where the functions for embossing and sharpening are encapsulated in their own respective `.cpp` files.

Delete `emboss_or_sharpen.cpp`, and copy the contents of `SummerSchoolLabSession/emboss_or_sharpen_multi_compilation_units` to your home directory:

```
-bash-3.2$ rm emboss_or_sharpen.cpp
-bash-3.2$ cp SummerSchoolLabSession/emboss_or_sharpen_multi_compilation_units/* .
```

This consists of a `.cpp` and `.h` file for each of the filter functions, and a `main.cpp` file which attempts to offload image filtering using a sievethread block and function pointers, as we have already seen. Have a quick look at the files to understand what's going on.

Now, after deleting the `outputc` directory from your previous build, compile everything:

```
-bash-3.2$ wine CellSieveCPP main.cpp emboss_pixel.cpp sharpen_pixel.cpp image.cpp
```

Then navigate to the `outputc` directory and type `make`. You should get two link errors:

```
/tmp/ccwkJ09u.o:(.data+0x10): undefined reference to
`_Z12emboss_pixelPU7__outerK3rgbiiEU3_SL1'
/tmp/ccwkJ09u.o:(.data+0x1c): undefined reference to
`_Z13sharpen_pixelPU7__outerK3rgbiiEU3_SL1'
```

The Cell linker is complaining that it can't find the functions `emboss_pixel` or `sharpen_pixel` for the SPEs (the error message displays the *mangled* names for these functions – notice that `__outer` forms part of the mangled name).

The trouble is that, since the compiler process each compilation unit in isolation, it does not have access to the source code for `emboss_pixel` or `sharpen_pixel` when it compiles `main.cpp`, which refers to these functions.[1]

The solution is to mark the prototypes of these functions with an *attribute* to specify that they should be duplicated with a particular outer pointer signature. In `emboss_pixel.h`, add:

```
__attribute__((__duplicate (rgb (const __outer rgb *, int, int))))
```

after the prototype for function `emboss_pixel`, before the semicolon which terminates the prototype. This attribute says to the compiler: "duplicate `emboss_pixel` as if it had been declared as **`rgb emboss_pixel(const __outer rgb* pixels, int i, int j);`**". Do the same in `sharpen_pixel.h`. You should now find that the application compiles, links and runs without any problems.

NOTE: in this version of the filters, `emboss_pixel` calls an auxiliary function, `greyscale`, which computes the floating point greyscale value for a pixel. Although we had to mark `emboss_pixel` with a duplication attribute, it was *not necessary* to also mark the `greyscale` function with this attribute: the compiler knows to duplicate `emboss_pixel` with `pixels` as an outer pointer, therefore when `pixels` is passed as the first parameter to `greyscale` the compiler knows it must duplicate `greyscale` with this parameter as an outer pointer. Thus it is only necessary to annotate the interface between compilation units with duplication attributes.

---

1 In this case, you might think that since we have passed all our `.cpp` files to the compiler at once, it should be able to find the relevant functions. This could be implemented as a special case, but in general we would like to be able to compile these files completely separately, using a makefile.

Optimising the sharpen filter with vector instructions

To finish our work with the image filter functions, we will see how a) generic Alti-Vec vector instructions can be used to optimise the sharpen filter for both PPE and SPE, and b) SPE-specific vector instructions can be used to further optimise the filter on SPEs.

Generic Alti-Vec vector instructions

Delete all `.cpp` and `.h` files except `image.cpp` and `image.h` from your home directory. Copy `SummerSchoolLabSession/emboss_or_sharpen_vectorized/emboss_or_sharpen_vectorized.cpp` to your home directory, and open this file.

For simplicity, we have gone back to a version of the filters which uses one compilation unit and does not use function pointers.

You will see that the `sharpen_pixel` function now declares two vector constants, `minus1` and `five`. Instead of multiplying each colour component of a pixel by either -1 or 5, the filter now interprets each pixel as a vector, and multiplies a whole pixel by either `minus1` or `five`. This potentially optimises the code since a vector multiply can be performed as a single instruction (we say *potentially* since the compiler may be clever enough to automatically perform this vectorization).

The last line of `sharpen_pixel` uses a "nasty cast" to turn `dest`, a `vector float`, back into something with type `rgb`. This relies on the fact that the `rgb` data type consists of four `float` values, thus has the same memory layout as a `vector float`.

The `vector float` type is part of the Alti-Vec language extensions, which also overloads the multiplication operator to work on vectors. These language extensions are supported for both the PPE and SPE, so this single version of `sharpen_pixel` works for both processors.

If you have time, compile and run this example to check that it still performs the sharpen filter correctly.

Using SPE-specific vector instructions

There is room for further improvement – the statements:

```
dest[0] = dest[0] > 1.0f ? 1.0f : (dest[0] < 0.0f ? 0.0f : dest[0]);
dest[1] = dest[1] > 1.0f ? 1.0f : (dest[1] < 0.0f ? 0.0f : dest[1]);
dest[2] = dest[2] > 1.0f ? 1.0f : (dest[2] < 0.0f ? 0.0f : dest[2]);
```

have the effect of "clamping" the result pixel so that each colour component is in the range [0, 1]. This clamping is done separately for each colour component, and can be vectorized to work on all components simultaneously.

To implement this optimization, we will use two SPE instructions, `spu_cmpgt` (compare greater than) and `spu_sel` (select). Here is a description of each:

- `spu_cmpgt(vector1, vector2)` – compares `vector1` and `vector2` element-wise, and returns a boolean vector, with position *i* set to true if and only if `vector1[`*i*`] > vector2[`*i*`]`
- `spu_sel(bool_vector, false_result, true_result)` – returns a vector with position *i* set to `false_result[`*i*`]` if and only if `bool_vector[`*i*`]` is false, and position *i* set to `true_result[`*i*`]` otherwise.

Example: let v = (1, 2, 3, 4) and w = (4, 3, 2, 1). Then if b is set to `spu_cmpgt`(v, w) we have b = (F, F, T, T). Then if we set u to `spu_sel`(b, w, v) we have u = (4, 3, 3, 4).

Copy and paste the code for `sharpen_pixel` to make a duplicate of this function. Change the function signature to:

```
sievethread rgb sharpen_pixel(const __outer rgb * pixels, int i, int j)
```

By marking the duplicated function with `sievethread`, and adding the `__outer` qualifier to `pixels`, we ensure that this version of the function is compiled only for the SPEs, to be called with a PPE pointer as the first parameter.

Now modify the body of the function to use the `spu_cmpgt` and `spu_sel` instructions to perform pixel clamping more efficiently. It's not too hard, but if you get stuck then look at `SummerSchoolLabSession/emboss_or_sharpen_vectorized_spe/emboss_or_sharpen_vectorized_spe.cpp`.

Compile and run your program to check that the sharpen filter still produces the correct result.

The drawback to this optimization is that we have made the code less portable. However, at least we could apply the optimization incrementally, first offloading a general version of the image sharpening filter to SPEs, then gradually modifying it to produce an SPE-optimized version.

Part 2 – Overloading and type inference

Although we haven't seen too many uses of the `__outer` keyword (which is a good thing!), outer pointers are at the heart of the Sieve Threads approach, whether declared explicitly or inferred. We'll now take a look at some issues with overloading and type inference related to outer pointers.

Overloading: intersecting circles

Copy `SummerSchoolLabSession/circles/circles.cpp` to your home directory. This file includes a method which takes two references to circles (consisting of x and y coordinates and a radius), and determines whether they intersect using a standard formula.

The `main` method in `circles.cpp` expects 12 command-line arguments – the centres and radii for four circles. Intersection is then checked for a selection of these circles, both inside and outside a sievethread block.

This example illustrates the fact that `__outer` applies to references (declared via `&` in C++) as well as pointers, and shows method duplication in action – each call to `show_intersection` requires a different variant of the method to be compiled:

```
show_intersection(c1, c2); // Use version compiled for PPE

sievethread
{
  Circle c3 = { x3, y3, r3 };
  Circle c4 = { x4, y4, r4 };
  show_intersection(c1, c2); // Compile for SPE: bool (__outer Circle&, __outer  Circle&)
  show_intersection(c3, c4); // Compile for SPE: bool (Circle&, Circle&)
  show_intersection(c3, c1); // Compile for SPE: bool (Circle&, __outer Circle&)
  show_intersection(c2, c4); // Compile for SPE: bool (__outer Circle&, Circle&)
};
```

Method duplication works well – an example run of the program shows that an appropriately compiled version of `show_intersection` is called each time, try it out:

```
[root@(none) ~]# ./main 10 10 10 9 9 10 100 100 3 20 20 7000
Circles [inner/outer?](centre=(10, 10), radius=10) and [inner/outer?](centre=(9, 9),
radius=10) intersect
Circles [inner/outer?](centre=(10, 10), radius=10) and [inner/outer?](centre=(9, 9),
radius=10) intersect
Circles [inner/outer?](centre=(100, 100), radius=3) and [inner/outer?](centre=(20, 20),
radius=7000) intersect
Circles [inner/outer?](centre=(100, 100), radius=3) and [inner/outer?](centre=(10, 10),
radius=10) do not intersect
Circles [inner/outer?](centre=(9, 9), radius=10) and [inner/outer?](centre=(20, 20),
radius=7000) intersect
```

The printed annotation `[inner/outer?]` indicates that the called method does not know whether the particular Circle resides on the SPE or PPE.

It can sometimes be useful (for purposes of optimization) to specialise behaviour of a method depending on the configuration of outer pointers/references with which it is called. Thus the Sieve Partitioning System supports overloading on `__outer`.

To illustrate this, let us make duplicate versions of the `show_intersection` method which print message indicating the “outerness” of the `Circle` reference arguments.

Add the following method to `circles.cpp`:

```
sievethread void show_intersection(Circle& c1, Circle& c2)
{
```

```
  printf("Circles [inner](centre=(%d, %d), radius=%d) and [inner](centre=(%d, %d),
radius=%d) ", c1.x, c1.y, c1.radius, c2.x, c2.y, c2.radius);
  // As before
  ...
};
```

The differences from the original `show_intersection` method are: the inclusion of the `sievethread` keyword, which specifies that the method should be compiled only for the SPE and theferore that the reference parameters point to SPE local memory as they don't have the `__outer` qualifier, and the `printf` statement which replaces `[inner/outer?]` with `[inner]`.

Now when we run the program, the call to `show_intersection` which take two inner `Circle` references is singled out:

```
[root@(none) ~]# ./main 10 10 10 9 9 10 100 100 3 20 20 7000
Circles [inner/outer?](centre=(10, 10), radius=10) and [inner/outer?](centre=(9, 9),
radius=10) intersect
Circles [inner/outer?](centre=(10, 10), radius=10) and [inner/outer?](centre=(9, 9),
radius=10) intersect
Circles [inner](centre=(100, 100), radius=3) and [inner](centre=(20, 20), radius=7000)
intersect
Circles [inner/outer?](centre=(100, 100), radius=3) and [inner/outer?](centre=(10, 10),
radius=10) do not intersect
Circles [inner/outer?](centre=(9, 9), radius=10) and [inner/outer?](centre=(20, 20),
radius=7000) intersect
```

Now make three more `sievethread` duplicates of `show_intersection`, overloaded with either one or both arguments as `__outer` references, such that `[outer]` or `[inner]` is printed depending on whether a `Circle` reference has the `__outer` qualifier or not. The output should look like this:

```
[root@(none) ~]# ./main 10 10 10 9 9 10 100 100 3 20 20 7000
Circles (centre=(10, 10), radius=10) and (centre=(9, 9), radius=10) intersect
Circles [outer](centre=(10, 10), radius=10) and [outer](centre=(9, 9), radius=10)
intersect
Circles [inner](centre=(100, 100), radius=3) and [inner](centre=(20, 20), radius=7000)
intersect
Circles [inner](centre=(100, 100), radius=3) and [outer](centre=(10, 10), radius=10) do
not intersect
Circles [outer](centre=(9, 9), radius=10) and [inner](centre=(20, 20), radius=7000)
intersect
```

If you get stuck, look at `SummerSchoolLabSession/circles_overloaded`.

Obviously in this case the overloading is illustrative but otherwise pointless. However, in practice it can be useful to overload a method to do a target-specific data movement when a particular pointer has the `__outer` qualifier.

Type inference: summing a list of type `float` *or* `double`

We shall now look at an example where method duplication fails.

Copy `SummerSchoolLabSession/sum_list/sum_list.cpp` to your home directory and open it.

This rather contrived program consists of a function which takes a list (as an array) of unknown type, and an argument specifying whether the elements have type `double` or `float`. The code uses pointer arithmetic and type casts to sum the list, assigning the result to the target of pointer result.

Compile the example with *CellSieveCPP*. You should get this error message:

```
*** ERROR: (Sieve1101) Cannot convert source type in assignment from 'char __outer*' into
'char *' while compiling duplicated function 'void sum_unknown_list(void __outer*, void
*, int , int );
'.
--- In file: sum_list.cpp, at line: 15, column: 2
          c$urrent = (char*)list;
```

The problem is that `sum_unknown_list` is called at line 49 with `float_list` as its first parameter. Since `float_list` is declared globally, it is an outer pointer. Thus the compiler tries to duplicate `sum_unknown_list` with signature: **`void sum_unknown_list(void __outer* , void* , int, int)`**, as indicated in the error message.

To do this duplication, essentially the compiler pretends that the list parameter of `sum_unknown_list` had been declared with type `void __outer*` (this is similar to template instantiation). In this case, the method would look like this:

```
void sum_unknown_list(void* __outer list, void* result, int size, int type)
{
          char* current;
          current = (char*)list;  // ERROR - list is an __outer pointer, current is not
  ...
```

The compiler rejects the assignment of `current` to `list`, since `list` is an `__outer` pointer and `current` is not. Rejecting such assignments is important, as we do not want to assign PPE memory locations to SPE pointers.

However, if we modify this code so that `current` is assigned to `list` *on declaration*:

```
void sum_unknown_list(void* __outer list, void* result, int size, int type)
{
   char* current = (char*)list;  // OK - current is given __outer qualifier by inference
```

then the error message goes away. (This modification is applied in `SummerSchoolLabSession/sum_list_working`.) This is because the compiler infers that `current` must have the `__outer` qualifier, as it has been initialised to something which itself has the `__outer` qualifier.

Running this example should produce the following output:

```
[root@(none) ~]# ./main
Sum of floats: 55.000000
Sum of doubles: 150.000000
```

Note also that the example contains various casts of from outer pointers to non-outer pointers. The compiler uses another simple inference rule to allow these casts to succeed: if an expression has type `__outer T *` and is cast to type `U *` then the compiler replaces the cast type with `__outer U *`.

These simple examples of type inference mean that it can be very straightforward to offload large portions of existing code. Typically after enclosing a piece of code in a sievethread block there are some error messages related to outer pointers, most of which can be easily solved by moving pointer assignments to initialisers as in the above example.

It seems restrictive and somewhat non-intuitive that

```
char* current = (char*)list;
```

works, when

```
char* current;
current = (char*)list;
```

does not. Indeed, future versions of the Sieve Partitioning System are expected to include more sophisticated type inference so that the latter code is also accepted.

Can you think of examples where even sophisticated type inference will not be able to sensibly type a program due to the way in which inner/outer pointers are used? Also, why is complex type inference potentially unattractive to users?

Part 3 – Experiment further!

If you have any time remaining then feel free to experiment further with the Sieve Partitioning System. Have a look at the pdf files in `SummerSchoolLabSession/Documentation` – these files describe more features of the system which you might like to try out.

Uninstall the Sieve Partitioning System

Before you leave, please run:

```
-bash-3.2$ wine uninstall
```

and click `Uninstall` to remove the Sieve Partitioning System from your workspace.