# Lab – SMP Lab

## Projects and source code

Unless instructed by the instructor otherwise, all projects and source code are available on the server. Directory /usr/local/projects has two sub-directories, ARM and DSP. The source for this ARM sub-directory is in the ARM subdirectory in a subdirectory called SMP.

## Purpose

The purpose of this lab is to demonstrate how the SMP LINUX distributes threads between multiple cores and as a result speed up the processing of time sensitive application running on the 4 A15 of KeyStone II.

The default application is a typical signal processing fir filter algorithm. Fir filters can be easily partition between multiple threads. The program was structured such that it is very easy to replace the fir filter with any generic easy to partition application.

### Task 1: Copy and observe the source files

It is assumed that the file system is mounted to the EVM (NFS boot, setenv boot net) and that the file system is in location /opt/filesys/studentN where N is the student number.

From the VNC window log-in as your student name (user name studentN, password WsN where N is the student number) and run the initialization script file

* Source /usr/local/staudentStartScript.sh

In the studentN file system location make a new directory (if it does not exist already) and name it applications and then make a subdirectory smp\_test

* cd /opt/filesys/studentN
* sudo mkdir applications
* cd applications
* sudo mkdir smp\_test
* cd smp\_test

Note – if the filesystem is located in a different directory change the instructions accordingly. For example, if the file system is in directory /opt/filesys/studentN/mcsdk\_3\_14 then the sequence of instruction is

* cd /opt/filesys/studentN/mcsdk\_3\_14
* sudo mkdir applications
* cd applications
* sudo mkdir smp\_test
* cd smp\_test

Next you copy the four source files to the new directory:

* sudo cp /usr/local/projects/SMP/smp\_test.c .
* sudo cp /usr/local/projects/SMP/multithreads.h .
* sudo cp /usr/local/projects/SMP/application.c .
* sudo cp /usr/local/projects/SMP/application.h .

Last copy the three Makefiles to the new directory

* sudo cp /usr/local/projects/SMP/Makefile\_no\_optimization .
* sudo cp /usr/local/projects/SMP/Makefile\_O2\_optimization .
* sudo cp /usr/local/projects/SMP/Makefile\_full\_optimization .

Question

Assume you want or need to change the algorithm that runs on the A15.

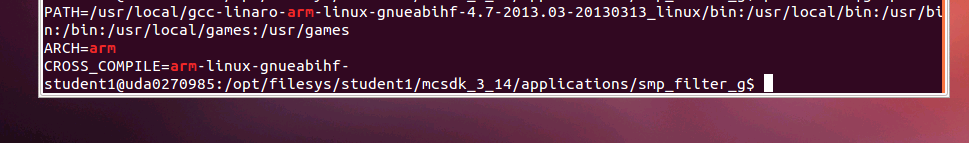
1. Does the file smp\_test.c need to be changed?
2. Does the include file multithreads.h needs to be changed?
3. Does the file application.c need to be changed or replaced?
4. Does the file application.h need to be changed or replaced?
5. Which instruction spans threads?
6. Optional – what does each of the parameters of the clone () function represent?

### Task 2: Compile and build the project, run it on a single core

From the VNC terminal you need to build the project. The path to the cross compiler should be define. To verify this do the following:

* sudo printenv |grep arm

The printing on the terminal will look like the following.



Make sure that the path to arm-linux-gnueabihf is defined and the arm-linux-gnueabihf is defined as the CROSS\_COMPIER. If not, run the initialization script:

* Source /usr/local/staudentStartScript.sh

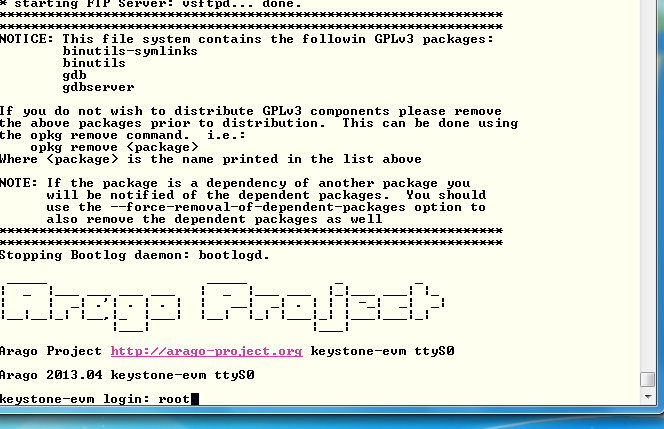
First build the project with full debug and no optimization:

* Make –f makefile\_No\_Optimization

Note that the optimization flag (dash capital O) is set to zero, and there is no –o (small o) thus the built project will have the default name a.out.

Before running the code you need to verify that the EVM is connected to the local network via Ethernet, which you have a terminal window (either Putty or Tera Term or other) into the EVM as explained in previous Labs, and that NSF boot from TFTP is working.

1. Power on the EVM
2. Wait for the login screen to come
3. Log as root:



1. Change the location of the terminal to the smp\_test directory:

* cd /applications/smp\_test
* ls

The 4 source files, the three Make files and the a.out file should be in the directory.

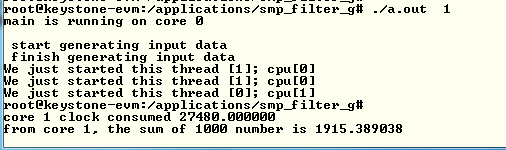
1. To run the code with a single thread first ensure that the file a.out has executable permission

* chmod +x a.out

1. Run the code

* ./a.out 1

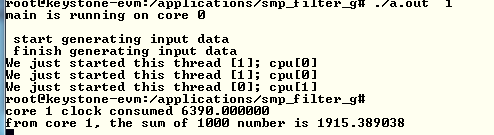
1. After a few seconds (wait until all the printing is done) the results should look similar to the follows:



1. Record the clock consumed value in the table below
2. Back in the VNC window re-build the executable with O2 optimization:

* Make –f makefile\_No\_Optimization

1. And run a.out again. The results should look like the following:



1. Back in the VNC window re-build the executable with full optimization:

* Make –f makefile\_full\_Optimization

1. And run a.out again.

Question

1. What is the speed up percentage of performance improvements when the optimization O2 is on?
2. What is the speed up percentage of performance improvements when the full optimization is on? - Compare to non-optimization, compare to –O2

### Task 3: Run the code on multiple cores

From this point on you only run the full optimized version of the a.out executable.

In this task you run the program on multiple cores. While the total processing time on all the cores remains almost the same, the elapsed time (the time that the slowest core consumes) will be reduced almost linearly by the number of cores that are involved.

The SMP operating system will distribute threads between cores. The number of thread in this program is limited to 32, but the number of cores is 4. So if the number of threads is bigger than 4, multiple threads will be assigned to each core.

Run the following cases and fill the table below:

* ./a.out 1
* ./a.out 2
* ./a.out 4
* ./a.out 8
* ./a.out 16
* ./a.out 32

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Number of threads | Core 0 total time consumed | Core 1 total time consumed | Core 2 total time consumed | Core 3 total time consumed | Slowest core time consumed | Total time consumed by the 4 cores |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Question

1. The shared ARM L2 cache is 4MB. Are all the vectors fit into the cache?
2. The private L1 cache for each A15 is 32KB. What is the maximum input vector that will not cause L1 cache to be trashed?
3. What is the speed up when 4 cores are used compare to a single core?
4. How does the cache size effect the total time when multiple threads are used?