

MCSDK Image Processing Demonstration Guide

Multicore Software Development Kit Image Processing Demonstration Guide

Last updated: 09/11/2015

Contents

Overview

Requirements

Software Design

- More about processing algorithms
- Framework for multicore
- Profiling of the algorithm
- User interface
- Software outline of the OpenMP demo
- Software outline of the Keystone II demo

Different Versions of Demo

- Software Directory Structure Overview
- Serial Code
 - Run Instructions for Serial based demo application
 - Build Instructions for Serial based demo application
- IPC-Based
 - Run Instructions for IPC based demo application
 - Build Instructions for IPC based demo application
- CToolsLib IPC-Based
 - Run Instructions for CToolsLib IPC based demo applications
 - CCS v5.4
 - Earlier CCS Versions
 - Expected Output of the CToolsLib Examples
 - Total Bandwidth
 - System Latency, System Bandwidth and Master Bandwidth
 - Event Profiler
 - Mem_PCT_WatchPoint
 - Mem_AETINT_WatchPoint
 - Statistical Profiling
- OpenMP-Based
 - Run Instructions for OpenMP based demo application
 - Build Instructions for OpenMP based demo application

Multicore System Analyzer integration and usage

Image Processing Demo Analysis with Prism

- Bring up the demo with Prism software
- What If* analysis

Multicore booting using MAD utilities

- Linking and creating bootable application image using MAD utilities
 - Pre-link bypass MAD image
- Booting the application image using IBL
 - Booting from Ethernet (TFTP boot)
 - Booting from NOR

Performance numbers of the demo

Overview

The Image Processing Demonstration illustrates the integration of key components in the Multicore Software Development Kit (MCSDK) on Texas Instruments (TI) multicore DSPs and System-on-Chips. The purpose of the demonstration is to provide a multicore software development framework on an evaluation module (EVM).

This document covers various aspects of the demonstration application, including a discussion on the requirements, software design, instructions to build and run the application, and troubleshooting steps. Currently, only SYS/BIOS is supported as the embedded OS.

This application shows implementation of an image processing system using a simple multicore framework. This application will run TI image processing kernels (a.k.a, *imagelib*) on multiple cores to do image processing (eg: edge detection, etc) on an input image.

There are three different versions of this demonstration that are included in the MCSDK. However, not all three versions are available for all platforms.

- *Serial Code*: This version uses file i/o to read and write image file. It can run on the simulator or an EVM target platform. The primary objective of this version of the demo is to run Prism and other software tools on the code to analyze the basic image processing algorithm.
- *IPC Based*: The IPC based demo uses SYS/BIOS IPC to communicate between cores to perform an image processing task parallel. See below for details.
- *OpenMP Based*: (Not available for C6657) This version of the demo uses OpenMP to run the image processing algorithm on multiple cores.

Note: The current implementation of this demonstration is not optimized. It should be viewed as the initial implementation of the BIOS MCSDK software eco-system for creating an image processing functionality. Further analysis and optimization of the demonstration are under progress.

Note: There are three versions of the demo provided in the release. The IPC based version runs on multiple cores and shows explicit IPC programming framework. The serial version of the demo runs on the simulator. The OpenMP version uses OpenMP to communicate between cores to process the input images. Unless explicitly specified, the IPC based version is assumed in this document.

Note: Before running the demo, please follow the software Getting Started Guide to configure your EVM properly.

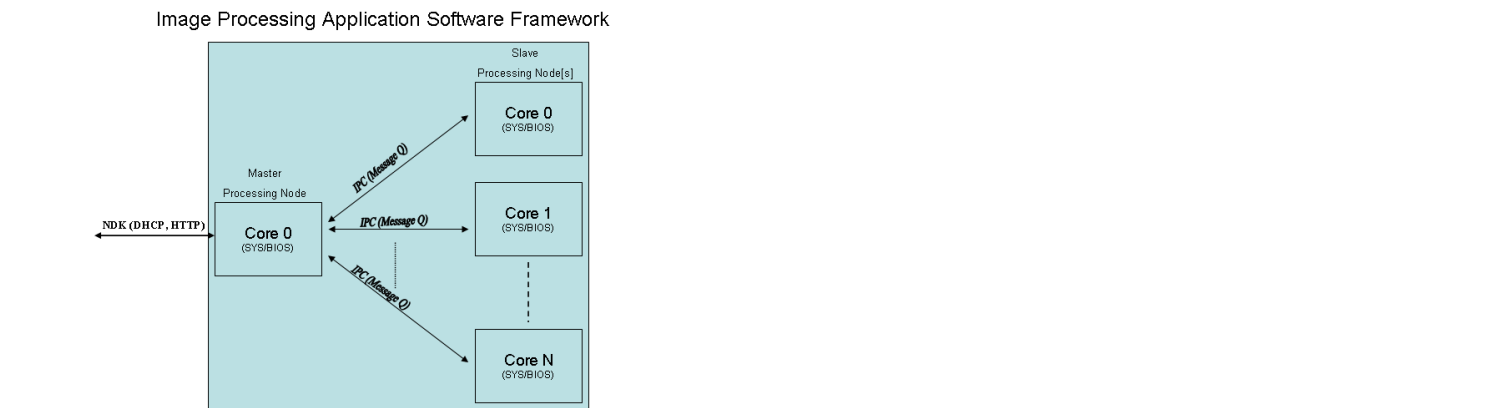
Requirements

The following materials are required to run this demonstration:

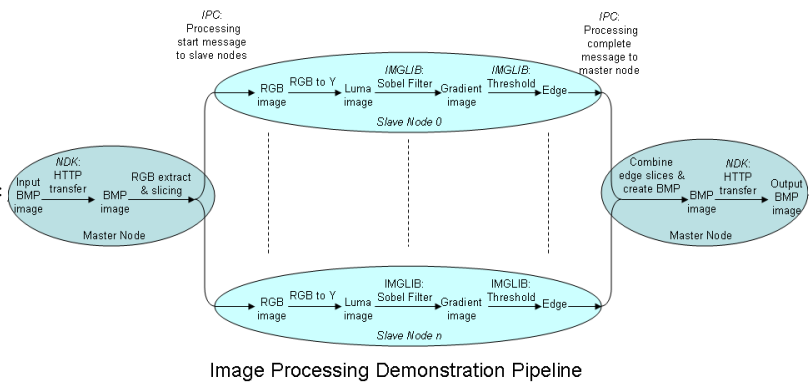
- TMS320C6x low cost EVMs [Check Image Processing release notes for supported platforms]
- Power cable
- Ethernet cable
- Windows PC with CCSv5

Software Design

The following block diagram shows the framework used to implement the image processing application:



The following diagram shows the software pipeline for this application:



More about processing algorithms

The application will use imagelib APIs for its core image processing needs.

Following steps are performed for edge detection

- Split input image into multiple overlapping slices
- If it is a RGB image, separate out the Luma component (Y) for processing (See YCbCr (<http://en.wikipedia.org/wiki/Ycbcr>) for further details)
- Run Sobel operator (http://en.wikipedia.org/wiki/Sobel_operator) (IMG_sobel_3x3_8) to get the gradient image of each slices
- Run the thresholding operation (IMG_thr_le2min_8) on the slices to get the edges
- Combine the slices to get the final output

Framework for multicore

The current framework for multicore is either IPC Message Queue based framework or OpenMP. Following are the overall steps (the master and threads will be run on 1 or more cores)

- The master thread will preprocess the input image (described in User interface section) to make a gray scale or luma image
- The master thread signal each slave thread to start processing and wait for processing complete signal from all slave threads
- The slave threads run edge detection function (described above) to generate output edge image of the slice
- Then the slave threads signal master thread indicating the processing completed
- Once master thread receives completion signal from all threads it proceeds with further user interface processing (described in User interface section)

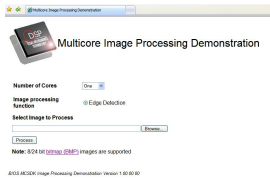
Profiling of the algorithm

- The profiling information live processing time will be presented at the end of the processing cycle
- Core image processing algorithms is instrumented using UIA for analysis and visualization using MCSA (Multicore System Analyzer)

User interface

The user input image will be a BMP image. The image will be transferred to external memory using NDK (http). Following are the states describing application user interface and their interaction

- At the time of bootup the board will bring configure IP stack with static/dynamic IP address and start a HTTP server
- The board will print the IP address in CCS console
- The user will use the IP address to open the index/input page (see link [Sample Input Page](#))
- The application will support BMP image format
- The master thread will extract the RGB values from BMP image
- Then the master thread will initiate the image processing (as discussed above) and wait for its completion
- Once the processing completes, it will create output BMP image
- The master thread will put input/output images in the output page (see link [Sample Output Page](#))



Software outline of the OpenMP demo

- The main task is called by OpenMP in core 0, spawns a task to initialize NDK, then gets/sets IP address and starts a web service to transfer user inputs and images. The main task then creates a mailbox and waits on a message post to the mailbox.
- The NDK calls a callback function to the application to retrieve the image data from user. The function reads the image and posts a message with the image information to the main task. Then it waits on a mailbox for a message post from main task.
- After receiving the message, the main task extracts RGB, splits the image into slices and processes each slices in different cores. A code snippet of this processing is provided below.

```
<syntaxhighlight lang="c">

1. pragma omp parallel for shared(p_slice, number_of_slices, ret_val) private(i)
for (i = 0; i < number_of_slices; i++) {

    DEBUG_PRINT(printf("Processing slice # %d\n", i));
    /* Process a slice */
    process_rgb (&p_slice[i]);
    if (p_slice[i].flag != 0) {
        printf("mc_process_bmp: Error in processing slice %d\n", i);
    }

    1. pragma omp atomic

    ret_val = -1;
}
DEBUG_PRINT(printf("Processed slice # %d\n", i));

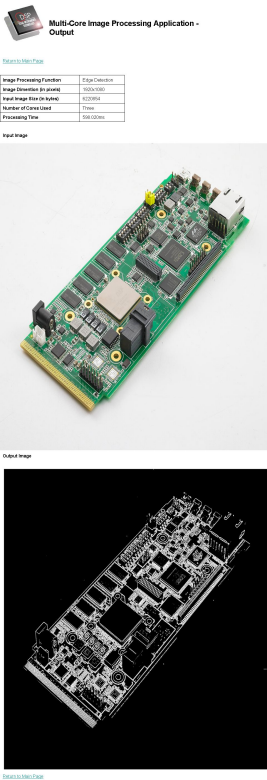
}

if (ret_val == -1) {

    goto close_n_exit;

}
}</syntaxhighlight>
```

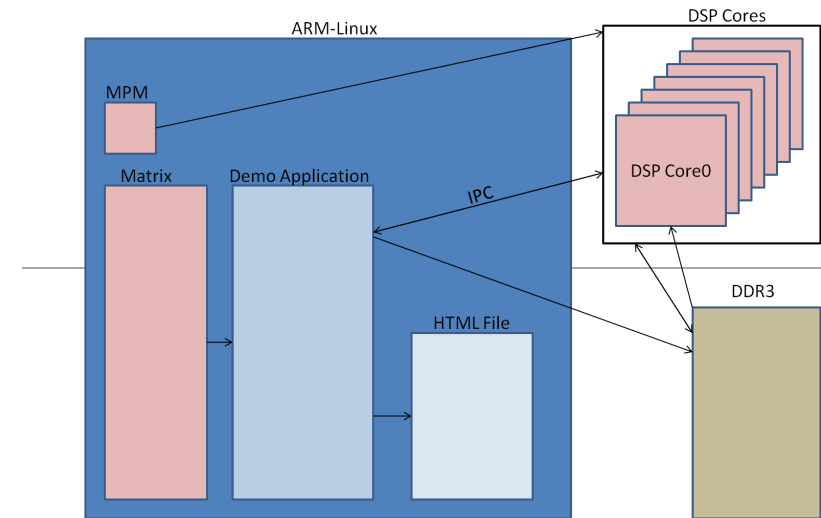
- After processing is complete, the main task creates the output image and sends the image information to the callback task using a message post. Then it waits on the mailbox again.
- The NDK callback task wakes up with the message post and sends the result image to the user.



Software outline of the Keystone II demo

The Image Processing Demo for Keystone II differs in the following ways from the original Image Processing Demo:

- The main process runs on the ARM cluster and is launched using the Matrix Application launcher.
- NDK is removed, as the web server is now run via Matrix.
- The DSP images are loaded using MPM.
- DDR3 memory is allocated by the ARM through requests to the DSP core0. The physical address returned by core0 is then mapped to user space.
- A diagram of the Keystone II demo framework is shown below:



Different Versions of Demo

Software Directory Structure Overview

The Image Processing Demonstration is present at <MCSDK INSTALL DIR>\demos\image_processing

- <MCSDK INSTALL DIR>\demos\image_processing\ipc\common directory has common slave thread functions which runs on all cores for the IPC based demo; The image processing function runs in this slave thread context
- <MCSDK INSTALL DIR>\demos\image_processing\ipc\master directory has main thread, which uses NDK to transfer images and IPC to communicate to other cores to process the images
- <MCSDK INSTALL DIR>\demos\image_processing\ipc\slave directory has the initialization function for all slave cores

- <MCSDK INSTALL DIR>\demos\image_processing\openmp\src directory has the main thread, which uses NDK to transfer images and OpenMP to communicate between cores to process the image
- <MCSDK INSTALL DIR>\demos\image_processing\ipc\evmc66###\master\slave directories have the master and slave CCS project files for the IPC based demo
- <MCSDK INSTALL DIR>\demos\image_processing\openmp\evmc66### directory has the CCS project files for the OpenMP based demo
- <MCSDK INSTALL DIR>\demos\image_processing\#####evmc66###\platform directory has the target configuration for the project
- <MCSDK INSTALL DIR>\demos\image_processing\serial directory has the serial version of the implementation
- <MCSDK INSTALL DIR>\demos\image_processing\utils directory has utilities used on the demo, like MAD config files
- <MCSDK INSTALL DIR>\demos\image_processing\images directory has sample BMP images

Serial Code

Run Instructions for Serial based demo application

The pre-compiled libraries are provided as a part of MCSDK release.

Please follow the procedures below to load images using CCS and run the demo.

Please refer the hardware setup guide for further the setup details.

- Connect the board to a Ethernet hub or PC using Ethernet cable.
- The demo runs in Static IP mode if User Switch 1 (SW9, position 2) is OFF else if it is ON then it runs DHCP mode. See the [TMDXEVM6678L EVM Hardware Setup](#) for the location of User Switch 1.
- If it is configured in static IP mode, the board will come up with IP address 192.168.2.100, GW IP address 192.168.2.101 and subnet mask 255.255.254.0
- If it is configured in DHCP mode, it would send out DHCP request to get the IP address from a DHCP server in the network.
- There is one image to be loaded on core 0. The image name is <MCSDK INSTALL DIR>\demos\image_processing\serial\Debug\image_processing_serial_simc6678.out.
- Connect the debugger and power on the board.
- It should open debug perspective and open debug window.
- Connect to only core 0, if the board is in no-boot mode make sure gel file is run to initialize ddr.
- Load image_processing_serial_simc6678.out to core 0.
- Run the core 0, in the CIO window, the board should print IP address information (eg: Network Added: If-1:192.168.2.100)
- Open a web browser in the PC connected to the HUB or the board.
- Enter the IP address of the board, it should open up the image processing demo web page.
- Please follow the instructions in the web page to run the demo.
- Note that sample BMP images are provided in <MCSDK INSTALL DIR>\demos\image_processing\images

Build Instructions for Serial based demo application

Please follow the steps below to re-compile the Serial based demo image (These steps assume you have installed the MCSDK and all dependent packages).

- Open CCS->Import Existing... tab and import project from <MCSDK INSTALL DIR>\demos\image_processing\serial.
- It should import image_processing_serial_simc6678 project.
- The project should build fine for Release and Debug profile.

IPC-Based


Run Instructions for IPC based demo application

The pre-compiled libraries are provided as a part of MCSDK release.

Please follow the procedures below to load images using CCS and run the demo.

Please refer the hardware setup guide for further the setup details.

- Connect the board to a Ethernet hub or PC using Ethernet cable.
- The demo runs in Static IP mode if User Switch 1 (SW9, position 2) is OFF else if it is ON then it runs in DHCP mode. See the [TMDXEVM6678L EVM Hardware Setup](#) for the location of User Switch 1.
- If it is configured in static IP mode, the board will come up with IP address 192.168.2.100, GW IP address 192.168.2.101 and subnet mask 255.255.254.0
- If it is configured in DHCP mode, it would send out DHCP request to get the IP address from a DHCP server in the network.
- There are two images to be loaded to master (core 0) and other cores. The core 0 to be loaded with <MCSDK INSTALL DIR>\demos\image_processing\ipc\evmc66###\master\Debug\image_processing_evmc66###_master.out image and other cores (referred as slave cores) to be loaded with <MCSDK INSTALL DIR>\demos\image_processing\ipc\evmc66###\slave\Debug\image_processing_evmc66###_slave.out image.
- Connect the debugger and power on the board.
- In CCS window, launch the target configuration file for the board.
- It should open debug perspective and open debug window with all the cores.
- Connect to all the cores and load image_processing_evmc66###_master.out to core 0 and image_processing_evmc66###_slave.out to all other cores.
- Run all the cores, in the CIO console window, the board should print IP address information (for eg: Network Added: If-1:192.168.2.100)
- Open a web browser in the PC connected to the HUB or the board.
- Enter the IP address of the board, it should open up the image processing demo web page.
- Please follow the instructions in the web page to run the demo.
- Note that, sample BMP images are provided in <MCSDK INSTALL DIR>\demos\image_processing\images

 **Note:** If you want to run the demo in static IP address mode, make sure the host PC is in same subnet or can reach the gateway. A sample setup configuration is shown below.

In Windows environment

Set up TCP/IP configuration of 'Wired Network Connection' as shown in [Wired Network Connection in Windows](#).

In Linux environment

Run following command to set the static IP address for the current login session on a typical Linux setup.

```
<syntaxhighlight lang="bash"> sudo ifconfig eth0 192.168.2.101 netmask 255.255.254.0 </syntaxhighlight>
```

Build Instructions for IPC based demo application

Please follow the steps below to re-compile the IPC based demo image (These steps assume you have installed the MCSDK and all the dependent packages).

- Open CCS->Import Existing... tab and import project from <MCSDK INSTALL DIR>\demos\image_processing\ipc.
- It should import two projects image_processing_evmc66##_master and image_processing_evmc66##_slave.
- Right click on each project->Properties to open up the properties window.
- Goto CCS Build->RTSC and check if in other repository have link to <MCSDK INSTALL DIR> (the actual directory).
- If IMGLIB C66x is unchecked, please select 3.0.1.0 to check it.
- The RTSC platform should have demos.image_processing.evmc66##.platform.
- The project should build fine.

CToolsLib IPC-Based

This demonstrates Common Platform Tracer (CPT) capabilities using image processing demo as an application. Please see below for details on the demo and steps to be followed to run it on the C6670/C6678/C6657 EVM. The instrumentation examples are provided on the IPC version of the demo. The following are the supported use cases:

1. System Bandwidth Profile
 1. Captures bandwidth of data paths from all system masters to the slave (or) slaves associated with one or more CP Tracers
 2. Demo: For the Image demo, instrument the external memory (MSMC & DDR3) bandwidth used by all system masters
2. System Latency Profile
 1. Captures Latency of data paths from all system masters to the slave (or) slaves associated with one or more CP Tracers
 2. Demo: For the Image demo, instrument the external memory (MSMC & DDR3) Latency values from all system masters
3. The CP tracer messages (system trace) can be exported out for analysis in 3 ways:
 1. System ETB drain using CPU (capture only 32KB (SYS ETB size in keystone devices) of system trace data)
 2. System ETB drain using EDMA (ETB extension to capture more than 32KB (SYS ETB size in keystone devices))
 3. External Emulator like XDS560 PRO or XDS560V2
4. Total Bandwidth Profile
 1. The bandwidth of data paths from a group of masters to a slave is compared with the total bandwidth from all masters to the same slave.
 1. The following statistics are captured:
 1. Percentage of total slave activity utilized by a selected group of masters
 2. Slave Bus Bandwidth (bytes per second) utilized by the selected group of masters
 3. Average Access Size of slave transactions (for all system masters)
 4. Bus Utilization (transactions per second) (for all system masters)
 5. Bus Contention Percentage (for all system masters)
 6. Minimum Average Latency (for all system masters)
 2. Demo: For Image Demo, compare DDR3 accesses by Core0 (master core) with the DDR3 accesses by all other masters (which includes Core1, Core2...)
5. Master Bandwidth Profile
 1. The bandwidth of data paths from two different group of masters to a slave is measured and compared.
 1. The following statistics are captured:
 1. Slave Bus Bandwidth (bytes per second) utilized by master group 0
 2. Slave Bus Bandwidth (bytes per second) utilized by master group 1
 3. Average Access Size of slave transactions (for both the master groups)
 4. Bus Utilization (transactions per second) (for both the master groups)
 5. Bus Contention Percentage (for both the master groups)
 6. Minimum Average Latency (for both the master groups)
 2. Demo: For Image Demo, compare DDR3 accesses by Core0 and Core1 with the DDR3 accesses by Core2 and Core3
6. Event Profile
 1. Capture new request transactions accepted by the slave. Filtering based on Master ID or address range is supported.
 1. The following statistics are captured for every new request event: Master ID which initiated this particular transaction
 1. Bus Transaction ID
 2. Read/Write Transaction
 3. 10 bits of the address (Address export mask selects, which particular 10 bits to export)
 2. Demo: For Image Demo, implement a Global SoC level watch point on a variable in DDR3 memory. Only Core0 is intended to read/write to this DDR3 variable. Unintended accesses by other masters (or Cores) should be captured and flagged.
7. Statistical profiling provides a light weight(based on the number of Trace samples needed to be captured) and coarse profiling of the entire application.
 1. PC trace is captured at periodic sampling intervals. By analyzing these PC trace samples, a histogram of all functions called in the application, with the following information:
 1. percent of total execution time
 2. number of times encountered
 2. Demo: In the Image Demo, Statistically profile the process_rgb() function, which processes a single slice of the Image.

Run Instructions for CToolsLib IPC based demo applications

The instructions to run each CToolsLib demo are the same as the non-CToolsLib demo; however, before pressing 'run' in CCS, you must perform the following steps:

CCS v5.4

If using CCS v5.4 or later, use the following instructions. Please see the next section for earlier versions of CCS.

For the event_profiler, master_bandwidth, system_bandwidth, system_latency, and total_bandwidth demos:

- While in the CCS Debug perspective, load all cores as you normally would with the IPC Demo.
- Go to **Tools->Hardware Trace Analyzer->Custom System Trace**.
- Select the appropriate **Transport Type** (depending on how the board is connected, either **560 V2 Trace** or **Pro Trace**, **ETB** is also acceptable).
- If using **560 V2 Trace** or **Pro Trace**, select the following options:
 - Buffer Type: **Stop-on-full**
 - Buffer Size: **64 MB**
 - Number of Pins: **4 pin**
 - Check **Synchronize trace collection with target run and halt**.
- Click **Start**.
- Run the demo.
- In the **Trace Viewer** tab, click **Stop**.
- For master_bandwidth, system_bandwidth, system_latency and total_bandwidth; a graphical representation of the results can be generated by clicking **Analyze->Memory Throughput** in the **Trace Viewer** tab.
- See the section below for details on the output of projects.

For the mem_aetint_watchpoint and mem_pct_watchpoint demos:

- While in the CCS Debug perspective, load all cores as you normally would with the IPC Demo.
- With the first core selected, go to **Tools->Hardware Trace Analyzer->Custom PC Trace**.
- Select the **Transport Type** as **ETB**, make sure **Synchronize trace collection with target run and halt** is selected and click **Start**.
- Run the demo.
- After the demo is complete, click **Stop** in the **Trace Viewer** tab.
- See the section below for details on the output of projects.

For the statistical_profiling demo:

- While in the CCS Debug perspective, load all cores as you normally would with the IPC Demo.
- Create a folder named 'temp' located at C:\temp.
- Run the program. (Note: The application will take considerably longer than normal).
- The output of the PC trace will be present at C:\temp\StatProf_etbdata.bin.

Earlier CCS Versions

If using a versions of CCS earlier than 5.4, use the following instructions.

For the event_profiler, master_bandwidth, system_bandwidth, system_latency, and total_bandwidth demos:

- While in the CCS Debug perspective; right click in the "Debug" window and click "show all cores."
- Under the "Non Debuggable Devices" section that appears, right click on CSSTM_0 and click "connect."
- Go to Tools->Trace Control.
- Click Receiver and choose 560 V2 Trace.
- In the CSSTM_0 tab, in the Trace Settings section, choose 4 pin for the Port width, 8 MB as the buffer size, Stop-on-full and check Synchronize with target.
- Click Apply and/or OK.
- Go to Tools->Trace Analyzer->Open Trace Connection in New View and choose the option that ends in CCSTM_0.
- Run the program and perform the demo as you normally would.
- After the demo has been run, pause the application to view the results.
- For master_bandwidth, system_bandwidth, system_latency and total_bandwidth; a graphical representation of the results can be generated by going to Tools->Trace Analyzer->Analysis->STM Analysis->STM Analysis Graph.
- See the section below for details on the output of projects.

For the mem_aetint_watchpoint and mem_pct_watchpoint demos:

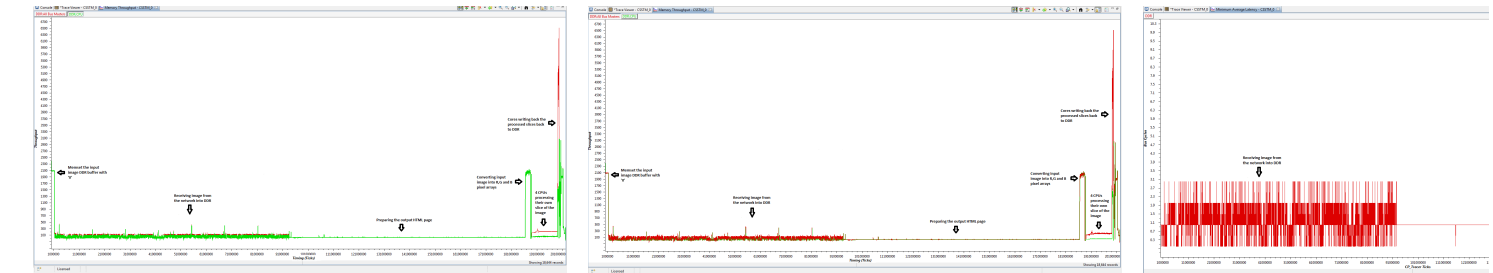
- Go to Tools->Trace Control.
- Select the C66x_0 tab, click receiver and choose ETB.
- Leave the values as default, and click Apply and/or OK.
- The Trace Display window should automatically open. Run the program.
- See the section below for details on expected output.

For the statistical_profiling demo:

- Create a folder named 'temp' located at C:\temp.
- Run the program. (Note: The application will take considerably longer than normal).
- The output of the ctools portion will be present at C:\temp\StatPof_etbdata.bin.

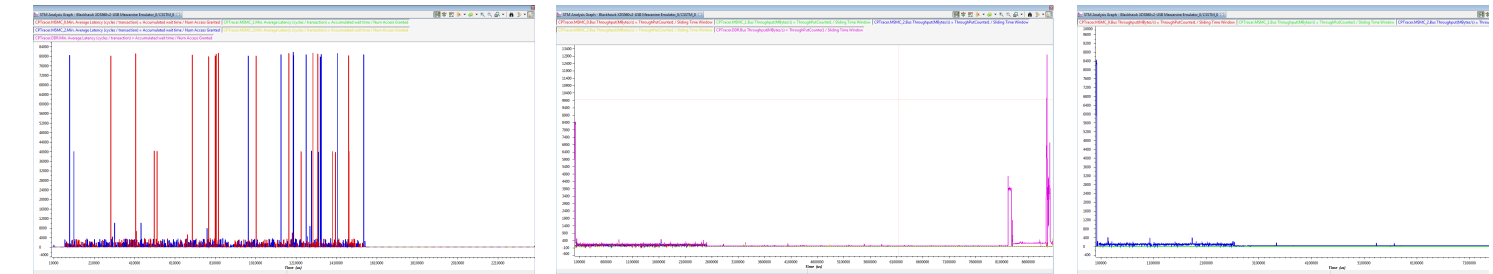
Expected Output of the CToolsLib Examples**Total Bandwidth**

Below are three images showing DDR3 bandwidth by core0 (master) and the system, as well as the DDR3 latency.



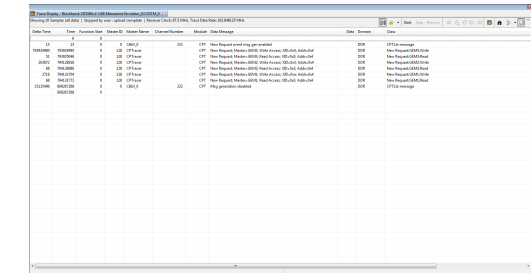
System Latency, System Bandwidth and Master Bandwidth

Below are the expected results for each example.



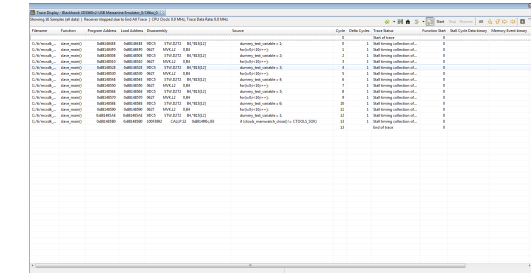
Event Profiler

Below is the expected result for the event_profiler example project. The project shows illegal accesses to a specific address by the slave cores (GEM1-GEM3).



Mem_PCT_WatchPoint

Below is the expected result for the mem_pct_watchpoint example project, which keeps watch on a dummy variable and counts the accesses to the variable.



Mem_AETINT_WatchPoint

Below is the expected result for the mem_aetint_watchpoint example project, which throws an exception and halts the master (core0) when a particular variable is accessed. The point in the code where the first unintended access happened can be viewed in the results.

[illegible]

For more information, please refer to the CTools UCLib documentation, available under [`<CTOOLSLIB INSTALL>\packages\ti\Tools_UCLib\doc\Tools_UCLib_html\index.html`].

This section we will use [Prism](http://www.criticalblue.com/prism/) software (<http://www.criticalblue.com/prism/>) to analyze the serial version of image processing demo. The steps below would assume Prism with C66x support is installed in the system and user completed the *Prism Getting Started - Tutorials* provided in the help menu.

Bring up the demo with Prism software

- Bring up the CCS as specified in the Prism documentation
- Edit *macros.ini* file from `<MCSDK INSTALL DIR>\demos\image_processing\serial` directory and change `../../../../imglib_c66x_#_#_#` to static path to IMGLIB package
- Open *Import Existing CCS Eclipse Project* and select search directory `<MCSDK INSTALL DIR>\demos\image_processing\serial`
- Check *Copy projects into workspace* and click *Finish*. This will copy the project to the workspace for Prism.
- Clean and re-compile the project
- Open the C6678 simulator, load the image to core0
- Open *Tools->GEL files*, select *Load GEL* and load/open *tisim_traces.gel* from `<CCSV5 INSTALL DIR>\ccs_base_####\simulation_csp_ny\env\ccs\import` directory
- Then select *CPU Register Trace->StartRegTrace* to start the trace, then run the program, wait till it finishes, then select *CPU Register Trace->StopRegTrace* to stop the trace
- Open *Prism->Show Prism Perspective*. It will start Prism Perspective
- Right click on the project and select *Create New PAD File*, it would open the New Prism Analysis Definition window, hit next
- It will open *Architecture Selection* window, select C6671 (single core) template. Then select *Finish* to open PAD file
- Select *Run->Debug Configurations->prismtrace*, this will convert the simulator generated traces to the traces required by Prism
- The PAD window should have filled in with default trace (PGSI, PGT) file names generated in above step
- Select *Read Trace* to read the trace
- After it read the trace, then select the complete trace from overview window and hit *Load Slice*
- The *Functions* tab will show the functions and their cycle information during the execution
- Observe the *Core 0* scheduling in the schedule window, you can place a marker for this run

What If analysis

The Prism tool allows user to analyze *What If* scenarios for the code

- *What If* the code is run on multiple cores
 - In the function window, right click on *process_rgb* function, select *Force Task* and hit *Apply*. This would make *process_rgb* function simulated as a separate task
 - In the *Architecture* tab, select *C6678 (8 Core)* template, hit *Apply*
 - Observe in *Schedule* tab, the change in execution when selected the *process_rgb* is simulated to run on 8 cores
 - A marker can be placed to compare the improvement
- *What If* the dependencies are removed
 - The *Dependencies* window helps to see and analyze the dependencies (which are preventing the task to be executed in multiple cores simultaneously)
 - Un-check *Serialized* check-boxes against dependency rows and hit *Apply*
 - Add the comparison marker in the *Schedule* tab and check the improvement

The Prism supports more functionality then described in this section. Please see Prism documentation for more information.

Multicore booting using MAD utilities

The detailed information on the Multicore Application Deployment a.k.a MAD utility is provided in [MAD user guide](http://processors.wiki.ti.com/index.php/MAD_Utils_User_Guide) (http://processors.wiki.ti.com/index.php/MAD_Utils_User_Guide) page.


This section will provide you the detail instructions on how the tool and boot the demo from flash/ethernet.

Linking and creating bootable application image using MAD utilities

The BIOS MCSDK installation provides MAD tool in `<MCSDK INSTALL DIR>\tools\boot_loader\mad-utils`. This package contains necessary tools to link the application to a single bootable image.

The image processing demo has following updates to create MAD image:

- The master and slave images are linked with `--dynamic` and `--relocatable` options.
- The MAD config files used to link the master and slave programs are provided in `<MCSDK INSTALL DIR>\demos\image_processing\utils\mad\evmc66###\config-files`. Following are few items to note on the config file.
 - *maptoolCfg_evmc#####.json* has the directory and file name information for the tools
 - *deployment_template_evmc#####.json* has the deployment configuration (it has device name, partition and application information). Following are some more notes on the configuration file.
 - For C66x devices, the physical address is 36 bits and virtual address is 32 bits for external devices, this includes MSMC SRAM and DDR3 memory subsystem.
 - The *secNamePat* element string is a regular expression string.
 - The sections *bss*, *neardata*, *rodata* must be placed in one partition and in the order it is shown here
- The build script `<MCSDK INSTALL DIR>\demos\image_processing\utils\mad\evmc66###\build_mad_image.bat` can be used to re-create the image

 **Note:** The compilation will split out lots of warning like *Incompatible permissions for partition ...*, it can be ignored for now. This is due to mis-match in partition permissions wrt. the sections placed in the partition

- The bootable image is placed in `<MCSDK INSTALL DIR>\demos\image_processing\utils\mad\evmc66###\images`

Pre-link bypass MAD image

Please see MAD user guide for more information on pre-link bypassed MAD image. The build script *build_mad_image_prelink_bypass.bat* can be used to build images with this mode.

Booting the application image using IBL

This image can be booted using IBL bootloader.

Following things to be noted on booting the image

- The image type/format is *ibl_BOOT_FORMAT_BBLOB*, so the IBL needs to be configured to boot this format
- The branch address (Branch address after loading) of the image [it is set to *0x9e001040* (or *0x80001040* if you are using BIOS MCSDK v 2.0.4 or prior) in MAL application], is different from default IBL boot address, so the IBL configuration needs to be updated to jump to this address

The following sections will outline the steps to boot the image from Ethernet and NOR using IBL. Please see IBL documentation on the detail information on booting.

Booting from Ethernet (TFTP boot)

- Change IBL configuration: The IBL configuration parameters are provided in a GEL file *<MCSDK INSTALL DIR>\tools\boot_loader\ibl\src\make\bin\i2cConfig.gel*. All the changes needs to be done in the function *setConfig_c66##_main()* of the gel file.
 - The IBL configuration file sets PC IP address 192.168.2.101, mask 255.255.255.0 and board IP address as 192.168.2.100 by default. If these address needs to be changed, open the GEL file, change *ethBoot.ethInfo* parameters in function *setConfig_c66##_main()*
 - Make sure the *ethBoot.bootFormat* is set to *ibl_BOOT_FORMAT_BBLOB*
 - Set the *ethBoot.blob.branchAddress* to *0x9e001040* (or *0x80001040* if you are using BIOS MCSDK v 2.0.4 or prior).
 - Note that the application name defaults to *app.out*

```
<syntaxhighlight lang="c"> menuitem "EVM c66## IBL";
```

```
hotmenu setConfig_c66##_main() {
```

```
    ibl.iblMagic = ibl_MAGIC_VALUE;
    ibl.iblEvmType = ibl_EVM_C66##L;
```

```
    ...
```

```
    ibl.bootModes[2].u.ethBoot.doBootp = FALSE;
    ibl.bootModes[2].u.ethBoot.useBootpServerIp = TRUE;
    ibl.bootModes[2].u.ethBoot.useBootpFileName = TRUE;
    ibl.bootModes[2].u.ethBoot.bootFormat = ibl_BOOT_FORMAT_BBLOB;
```

```
    SETIP(ibl.bootModes[2].u.ethBoot.ethInfo.ipAddr, 192,168,2,100);
    SETIP(ibl.bootModes[2].u.ethBoot.ethInfo.serverIp, 192,168,2,101);
    SETIP(ibl.bootModes[2].u.ethBoot.ethInfo.gatewayIp, 192,168,2,1);
    SETIP(ibl.bootModes[2].u.ethBoot.ethInfo.netmask, 255,255,255,0);
```

```
    ...
```

```
    ibl.bootModes[2].u.ethBoot.ethInfo.fileName[0] = 'a';
    ibl.bootModes[2].u.ethBoot.ethInfo.fileName[1] = 'p';
    ibl.bootModes[2].u.ethBoot.ethInfo.fileName[2] = 'p';
    ibl.bootModes[2].u.ethBoot.ethInfo.fileName[3] = '.';
    ibl.bootModes[2].u.ethBoot.ethInfo.fileName[4] = 'o';
    ibl.bootModes[2].u.ethBoot.ethInfo.fileName[5] = 'u';
    ibl.bootModes[2].u.ethBoot.ethInfo.fileName[6] = 't';
    ibl.bootModes[2].u.ethBoot.ethInfo.fileName[7] = '\0';
    ibl.bootModes[2].u.ethBoot.ethInfo.fileName[8] = '\0';
    ibl.bootModes[2].u.ethBoot.ethInfo.fileName[9] = '\0';
    ibl.bootModes[2].u.ethBoot.ethInfo.fileName[10] = '\0';
    ibl.bootModes[2].u.ethBoot.ethInfo.fileName[11] = '\0';
    ibl.bootModes[2].u.ethBoot.ethInfo.fileName[12] = '\0';
    ibl.bootModes[2].u.ethBoot.ethInfo.fileName[13] = '\0';
    ibl.bootModes[2].u.ethBoot.ethInfo.fileName[14] = '\0';
```

```
    ibl.bootModes[2].u.ethBoot.blob.startAddress = 0x9e000000 /*0x80000000 for BIOS MCSDK v2.0.4 or prior*/; /* Load start address */
    ibl.bootModes[2].u.ethBoot.blob.sizeBytes = 0x20000000;
    ibl.bootModes[2].u.ethBoot.blob.branchAddress = 0x9e001040 /*0x80001040 for BIOS MCSDK v2.0.4 or prior*/; /* Branch address after loading */
```

```
    ibl.chkSum = 0;
```

```
} </syntaxhighlight>
```

- Write IBL configuration:
 - Connect the board using JTAG, power on the board, open CCS, load the target and connect to core 0. Select *Tools->GEL Files* and in the GEL Files window right click and load GEL. Then select and load *<MCSDK INSTALL DIR>\tools\boot_loader\ibl\src\make\bin\i2cConfig.gel*.
 - Load I2C writer *<MCSDK INSTALL DIR>\tools\boot_loader\ibl\src\make\bin\i2cparam_0x51_c66##_le_0x500.out* to Core 0 and run. It will ask to run the GEL in console window. Run the GEL script from *Scripts->EVM c66##->setConfig_c66##_main*.
 - Open the CCS console window and hit enter to complete the I2C write.
- Booting the image:
 - Disconnect the CCS from board, power off the board.
 - Connect ethernet from board to switch/hub/PC and UART cables from board to PC.
 - Make sure your PC have the IP address specified above.
 - Set the board dip switches to boot from ethernet (TFTP boot) as specified in the hardware setup table (TMDXEVM6678L (http://processors.wiki.ti.com/index.php/TMDXEVM6678L_EVM_Hardware_Setup#Boot_Mode_Dip_Switch_Settings)) TMDXEVM6670L (http://processors.wiki.ti.com/index.php/TMDXEVM6670L_EVM_Hardware_Setup#Boot_Mode_Dip_Switch_Settings))
 - Copy the demo image *<MCSDK INSTALL DIR>\demos\image_processing\utils\mad\evmc66##\images\mcip-c66##_le.bin* to tftp directory and change its name to *app.out*
 - Start a tftp server and point it to the tftp directory

- Power on the board. The image will be downloaded using TFTP to the board and the serial port console should print messages from the demo. This will also print the configured IP address of the board
- Use the IP address to open the demo page in a browser and run the demo

Booting from NOR

- Change IBL configuration: The IBL configuration parameters are provided in a GEL file `<MCSDK INSTALL DIR>\tools\boot_loader\ibl\src\make\bin\i2cConfig.gel`. All the changes needs to be done in the function `setConfig_c66##_main()` of the gel file.
 - Make sure the `norBoot.bootFormat` is set to `ibl_BOOT_FORMAT_BBLOB`
 - Set the `norBoot.blob[0][0].branchAddress` to `0x9e001040` (or `0x80001040` if you are using BIOS MCSDK v 2.0.4 or prior)

```
<syntaxhighlight lang="c">
```

```
menuitem "EVM c66## IBL";
```

```
hotmenu setConfig_c66##_main() {
```

```
    ibl.iblMagic = ibl_MAGIC_VALUE;
    ibl.iblEvmType = ibl_EVM_C66##L;
```

```
    ...
```

```
    ibl.bootModes[0].bootMode = ibl_BOOT_MODE_NOR;
    ibl.bootModes[0].priority = ibl_HIGHEST_PRIORITY;
    ibl.bootModes[0].port = 0;
```

```
    ibl.bootModes[0].u.norBoot.bootFormat = ibl_BOOT_FORMAT_BBLOB;
    ibl.bootModes[0].u.norBoot.bootAddress[0][0] = 0; /* Image 0 NOR offset byte address in LE mode */
    ibl.bootModes[0].u.norBoot.bootAddress[0][1] = 0xA00000; /* Image 1 NOR offset byte address in LE mode */
    ibl.bootModes[0].u.norBoot.bootAddress[1][0] = 0; /* Image 0 NOR offset byte address in BE mode */
    ibl.bootModes[0].u.norBoot.bootAddress[1][1] = 0xA00000; /* Image 1 NOR offset byte address in BE mode */
    ibl.bootModes[0].u.norBoot.interface = ibl_PMEM_IF_SPI;
    ibl.bootModes[0].u.norBoot.blob[0][0].startAddress = 0x9e000000 /*0x80000000 for BIOS MCSDK v2.0.4 or prior*/; /* Image 0 load start address in LE mode */
    ibl.bootModes[0].u.norBoot.blob[0][0].sizeBytes = 0xA00000; /* Image 0 size (10 MB) in LE mode */
    ibl.bootModes[0].u.norBoot.blob[0][0].branchAddress = 0x9e001040 /*0x80001040 for BIOS MCSDK v2.0.4 or prior*/; /* Image 0 branch address after loading in LE mode */
```

```
    ...
```

```
    ibl.chkSum = 0;
```

```
}
```

```
</syntaxhighlight>
```

- Write IBL configuration:
 - Connect the board using JTAG, power on the board, open CCS, load the target and connect to core 0. Select *Tools->GEL Files* and in the GEL Files window right click and load GEL. Then select and load `<MCSDK INSTALL DIR>\tools\boot_loader\ibl\src\make\bin\i2cConfig.gel`.
 - Load I2C writer `<MCSDK INSTALL DIR>\tools\boot_loader\ibl\src\make\bin\i2cparam_0x51_c66##_le_0x500.out` to Core 0 and run. It will ask to run the GEL in console window. Run the GEL script from *Scripts->EVM c66##->setConfig_c66##_main*
 - Open the CCS console window and hit enter to complete the I2C write
- Write NOR image:
 - Copy application image (`<MCSDK INSTALL DIR>\demos\image_processing\utils\mad\evmc66##\images\mcip-c66##_le.bin`) to `<MCSDK INSTALL DIR>\tools\writer\nor\evmc66##\bin\app.bin`
 - Connect the board using JTAG, power on the board, open CCS, load the target and connect to core 0. Make sure the PLL and DDR registers are initialized from the platform GEL (if it is not done automatically, run *Global_Default_Setup* function from the GEL file). Load image `<MCSDK INSTALL DIR>\tools\writer\nor\evmc66##\bin\norwriter_evm66##.out`
 - Open memory window and load the application image (`<MCSDK INSTALL DIR>\demos\image_processing\utils\mad\evmc66##\images\mcip-c66##_le.bin`) to address `0x80000000`
 - Be sure of your **Type-size** choice 32 bits
 - Hit run for NOR writer to write the image
 - The CCS console will show the write complete message
- Boot from NOR:
 - Disconnect CCS and power off the board
 - Set the board dip switches to boot from NOR (NOR boot on image 0) as specified in the hardware setup table (TMDXEVM6678L (http://processors.wiki.ti.com/index.php/TMDXEVM6678L_EVM_Hardware_Setup#Boot_Mode_Dip_Switch_Settings) TMDXEVM6670L (http://processors.wiki.ti.com/index.php/TMDXEVM6670L_EVM_Hardware_Setup#Boot_Mode_Dip_Switch_Settings))
 - Connect ethernet cable from board to switch/hub
 - Connect serial cable from board to PC and open a serial port console to view the output
 - Power on the board and the image should be booted from NOR and the console should show bootup messages
 - The demo application will print the IP address in the console
 - Use the IP address to open the demo page in a browser and run the demo

Performance numbers of the demo

The following table compares the performance between OpenMP and explicit IPC based image processing demo applications.

Note: The results shown were taken during the BIOS-MCSDK 2.1.0 beta, results taken with current component versions may vary.

	Processing time for a ~16MB BMP image (in msec)	
Number of Cores	OpenMP based demo	Explicit IPC based demo
1	290.906	290.378
2	150.278	149.586
3	101.697	100.75
4	77.147	77.485
5	63.154	63.318
6	54.709	54.663
7	49.144	47.659
8	42.692	42.461

The performance numbers seem to be similar in both cases. This might be due to the nature of the demo application. It spends most of its processing time on actual image processing and sends, at most, 16 IPC messages between cores. So the contribution of the communication delays (IPC vs. OMP/IPC) are very minimal compared to any significant difference in processing times.

	Keystone=							
{{		■ For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum	C2000= <i>For technical support on the C2000 please post your questions on The C2000 Forum.</i> Please post only comments about the article MCSDK Image Processing Demonstration Guide here.	DaNinci= <i>For technical support on DaVinciplease post your questions on The DaVinci Forum.</i> Please post only comments about the article MCSDK Image Processing Demonstration Guide here.	MSP430= <i>For technical support on MSP430 please post your questions on The MSP430 Forum.</i> Please post only comments about the article MCSDK Image Processing Demonstration Guide here.	OMAP35x= <i>For technical support on OMAP please post your questions on The OMAP Forum.</i> Please post only comments about the article MCSDK Image Processing Demonstration Guide here.	OMAPL1= <i>For technical support on OMAP please post your questions on The OMAP Forum.</i> Please post only comments about the article MCSDK Image Processing Demonstration Guide here.	MAVRK= <i>For technical support on MAVRK please post your questions on The MAVRK Toolbox Forum.</i> Please post only comments about the article MCSDK Image Processing Demonstration Guide here.
1. switchcategory:MultiCore=		■ For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum	Please post only comments related to the article MCSDK Image Processing Demonstration Guide here.					

Retrieved from "[https://processors.wiki.ti.com/index.php?title=MCSDK Image Processing Demonstration Guide&oldid=206560](https://processors.wiki.ti.com/index.php?title=MCSDK_Image_Processing_Demonstration_Guide&oldid=206560)"

This page was last edited on 11 September 2015, at 08:29.

Content is available under Creative Commons Attribution-ShareAlike unless otherwise noted.