

TI Designs

Monte-Carlo Simulation on AM57x Using OpenCL to Leverage DSP Acceleration



TI Designs

TI Designs provide the foundation that you need including methodology, testing and design files to quickly evaluate and customize the system. TI Designs help *you* accelerate your time to market.

Design Resources

TIDEP0046	Tool Folder Containing Design Files
AM5728	Product Folder
TMDXEVM5728	Product Folder
PROCESSOR-SDK-AM57X	Product Folder



[ASK Our E2E Experts](#)
[WEBENCH® Calculator Tools](#)

Design Features

- Enables Use of the DSP Accelerators Without Requiring the User to Have Expert Knowledge of DSP
- Provides an Example of the Monte-Carlo Algorithm to Generate Gaussian Random Sequences That Run Faster on the C66x DSP Than on The ARM® Cortex®-A15 Core
- Offers a Complete System Reference Design With Example Software Implemented and Tested Using the TI Processor SDK and the TI AM57x EVM
- Includes Software Source, Schematics, Bill of Materials, and Design Files
- Applicable to Any Application That Uses the Monte-Carlo Simulation.

Featured Applications

- Business Strategy
- Radio Channel Simulation
- Personal Finance
- Traffic Load (Road Congestion, Network Capacity, and More)



An IMPORTANT NOTICE at the end of this TI reference design addresses authorized use, intellectual property matters and other important disclaimers and information.

ARM, Cortex are registered trademarks of ARM Limited.
Neon is a trademark of ARM Limited.
Linux is a registered trademark of Linux.
All other trademarks are the property of their respective owners.

1 Design Summary

This TI Design is an example of how to leverage DSP acceleration through OpenCL. OpenCL makes using the DSP easy for developers and is used for applications such as medical imaging, currency counters and sorters, vision inspection systems, and others. This design shows DSP accelerators for the Monte-Carlo simulation using the Linux® OpenCL program that run on the ARM Cortex-A15 CPU. Monte-Carlo simulation is a commonly used tool in many fields such as physics, communications, public utilities, and financing. Generating long Normal (Gaussian) distributed random number sequence is an essential part of many Monte-Carlo simulations. The computational load to generate long normal distributed random sequence is substantial. In many models, this load consumes most of the CPU cycles. DSP accelerators are designed to efficiently execute digital signal algorithms, such as the generation of random sequence. This design uses the DSP to generate the random sequence using a standard OpenCL code running on the Cortex-A15 processor under the Linux operating system. [Figure 1](#) shows the AM57 EVM.



Figure 1. AM57 EVM

1.1 Introduction to Generating Normal Distributed Random Sequence

True Random Number Generation is generated by special hardware. TI's Security Accelerator IP has a true random number generation. Pseudo Random Number Generation (PRNG) can be generated by software and has limited randomness.

Several methods to generate uniformly distributed random sequence are available. A commonly use method is the Linear Congruential Generator (LCG) method. The LCG starts with an initial seed and generates random sequence based on [Equation 1](#):

$$X_{n+1} = (a \times X_n + c) \text{ Mod } (M) \quad (1)$$

The following are values and their descriptions:

- X₀**— a seed
- M**— the modulo
- a**— the multiplier
- c**— the increment

Under certain conditions, the length of the sequence (the number of random numbers before the sequence starts repeating) is M-1. Using [Equation 1](#) generates numbers that are pseudo-uniformly distributed between 1 and M. Scaling LCG sequence to a uniformly distributed sequence in the range of [-1, 1] is easy.

To convert uniformly distributed random sequence into a Normal Gaussian random sequence, the design uses the polar form of the Box-Muller transformation (see Reference 1 in [Section 4](#)).

The Normal Gaussian distribution has two parameters, the average, and the standard deviation.

For a sequence of N random variables:

$X(n), n = 1, \dots, N$

The average (or $E(X)$) $\mu = (\sum X(n)) \div N$

Standard Deviation $\sigma = \text{sqrt}[E(X - \mu)^2]$

For a standard normal distribution case, $\sigma = 0$ and $\mu = 1$.

Common algorithm to get Gaussian random variables from uniformly distributed random variable is as follows:

1. Get two uniformly distributed $(-1.0, 1.0)$, x and y
2. Calculate $w = (x^2 + y^2)$
3. If $w < \text{sqrt}(-2.0 \times \log(w)) \div w$
4. $y1 = x^2y$
5. $y2 = y^2x$

$Y1$ and $Y2$ are two normal distributed random values.

1.2 Randomness, Sequence Length, and Parallel Computation

Multiple tests are suggested in the literature for the “randomness” of a Normal Random sequence:

- Have the correct distribution ($\lim (1 \div n \times S(X(n))) \sim N(0,1)$)
- No predictability (After reaching the length of the sequence, the sequence begins to repeat. Longer sequences are preferable to shorter sequences.)
- Auto-correlation goes to infinite, crossing correlation goes to zero
- ($\lim (S(X(n) \times X(n)) \rightarrow \infty)$)
- ($\lim (s(X(n) \times X(n-k)) \rightarrow 0)$)

Measure the entropy—lack of order or predictability; For a finite sequence X , the entropy is defined in the formula in [Equation 1](#). (see the discussion in Reference 2 in [Section 4](#)).

$$H(X) = -\sum_{\chi} P[X = \chi] \log_2 P[X = \chi]$$

Figure 2. Equation 2

Many publications discuss how to choose the constants a , M , and c , and the initial seed x_0 to achieve positive randomness features (see Reference 3 in [Section 4](#)). TI chose a and M as constants for [Equation 1](#) to ensure a long pseudo random sequence.

For this TI design to use the full power of the multiple DSP accelerators, it must merge multiple independent random sequences into a single random sequence while preserving the positive randomness attributes. Reference 4 in [Section 4](#) suggests a method of choosing the additive constant c for parallel generation. This design uses a set of prime numbers for parallel generating of normal random sequence.

1.3 OpenCL Implementation

OpenCL is a portable heterogeneous standard computing language that supports the easy use of generic accelerators for parallel processing applications. When writing the OpenCL applications, knowledge about the architecture of the accelerator is unnecessary. This code can run on different devices with different accelerator architectures. Generic accelerator code (kernel) is written in C or other standard language, and the system knows what compiler to use to convert the code into an accelerator program. Accelerators control code is a generic OpenCL code that requires no knowledge of the architecture. If the developer is familiar with the accelerator architecture, and is willing to give up portability, architecture optimized executable may be used by the OpenCL system.

OpenCL standard body is KHRONOS. TI implementation of OpenCL on the AM57x is compliant with version 1.1, see .

In this TI design, the DSP accelerator code is developed using ANSI C language. Thus, the project may be easily ported to other devices with different accelerator architecture.

1.4 OpenCL Platform Model

Figure 3 shows the OpenCL Platform model.

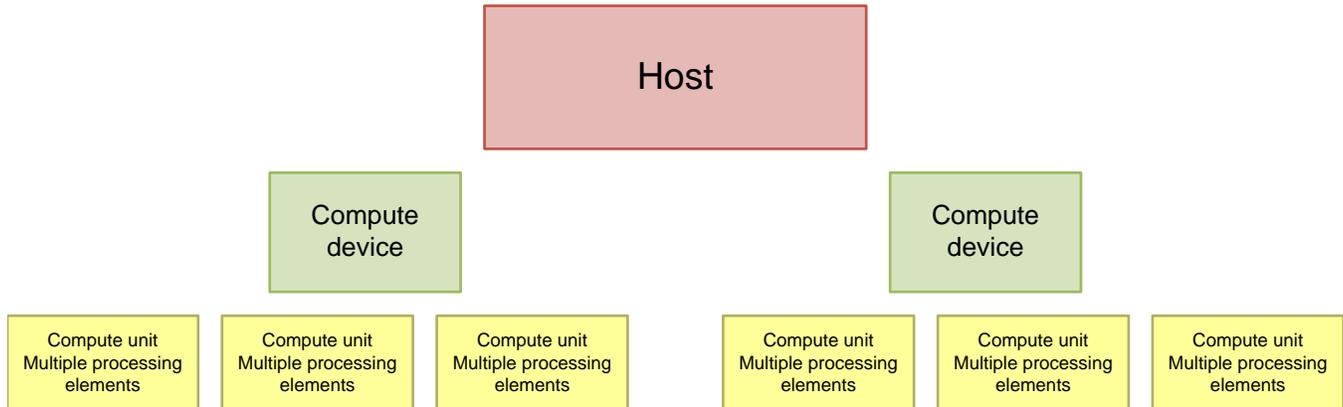


Figure 3. OpenCL Platform Model

A host is connected to one or more OpenCL devices. An OpenCL device is a collection of one or more compute units that share the same architecture. Compute units may have multiple processing elements. In the case of the AM57x, the host is the ARM Cortex-A15 cores running Linux SMP operating system. The compute device is the set of C66x DSPs, and the compute unit is a DSP core accelerator.

1.5 DSP Accelerator

The accelerators used in this TI design are C66x based. The kernel code is written in standard ANSI C, so no intrinsic or assembly language is used. The code takes advantage of the C66x memory architecture, and the eight functional units inside the core.

- The code uses TI real-time standard optimized Math library for standard Math functions such as square root, log, one over x, and so on. The runtime library that contains these functions is part of the standard release and is linked by the linker. Any device that supports C must have a similar library (with the same or very similar syntax) so porting these functions to another architecture is simple.
- The code takes advantage of the L1 Data SRAM part of the C66. This is a 32-KB area of zero wait-state access time that is used to store and retrieve intermediate values. Not all accelerators have L1 SRAM. The access to the memory is hard-coded in the DSP code.
- To use the DSP internal resources, each core generates two sequences that are later combined into one sequence. The algorithm follows Reference 4 in Section 4 to ensure randomness of the combined sequence.
- The same method used in Reference 4 in Section 4 is used to combine multiple DSP core sequences into a single sequence.

Figure 4 shows the C66x DSP block diagram.

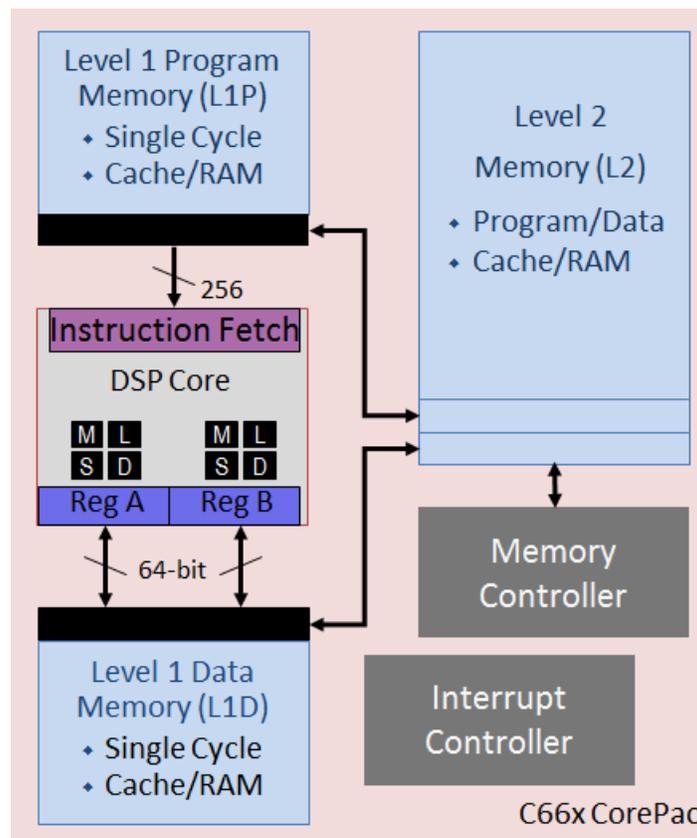


Figure 4. C66x Block Diagram

1.6 Kernel Code Design

OpenCL is available on a multitude of programming languages including C and C++. This TI design uses C++ for the OpenCL code, and ANSI C code for the accelerator-kernels. The accelerator kernel code may be presented as an ASCII string, or in separate C files. The compilation of the kernel string is done during run time by the system. The compilation of C files is done during the build process; however, linking the code and getting the executable from the compiled files are done during run time. This process increases the OpenCL runtime overhead; however, the OpenCL being portable to any OpenCL device is advantageous for the user. Other advantages for users are the benchmarking of applications over multiple devices, and the updating or scaling of the application.

2 Building Applications

2.1 Prepare the AM57x EVM Bootable SD

The TI processor SDK is a unified set of software building blocks that facilitate development of applications across multiple devices. The TI processor is in the public domain and can be loaded and used free of charge. [Figure 5](#) and [Figure 6](#) show part of the download page for the [PROCESSOR-SDK-LINUX-AM57X 02_00_01_07](#), see Reference 5 in the [Section 4](#) section.

PROCESSOR-SDK-LINUX-AM57X Product Downloads	
Title	Description
AM57xx Linux SDK Essentials	
ti-processor-sdk-linux-am57xx-evm-01.00.01.00-Linux-x86-Install.bin	AM57xx EVM Linux SDK
AM57xx Linux SDK Optional Addons	
Code Composer Studio 6.1.0	Link to Code Composer Studio 6.1.0
Download Pinmuxtool	AM57xx Pin Mux Configuration Utility (coming soon)
AM57xx Linux SDK SD Card Creation	
Linux SD Card Creation Wiki	Instructions for creating an SD Card with Linux
Windows SD Card Creation Wiki	Instructions for creating an SD Card with Windows
am57xx-evm-01.00.01.00.img.zip	Only used when creating an SD Card on Windows
AM57xx Linux SDK Individual Components (all of the below components are bundled within the Linux SDK Essentials package)	
Download Linaro Toolchain	Standalone Linaro Toolchain - Linaro GCC 4.7 2013.03 hard-float toolchain
am57xx-evm-sdk-src-01.00.01.00.tar.gz	AM57xx Linux SDK BSP Source Code
am57xx-evm-sdk-bin-01.00.01.00.tar.gz	AM57xx Linux SDK prebuilt BSP binaries and root filesystem
AM57xx Linux SDK Arago Source Tarball	

Figure 5. Processor-SDK-Linux AM57X Install Page

To burn an SD-bootable card using a Windows computer, follow the instructions found on the *Windows SD Card Creation Wiki* and the image to burn is in the zip file that follows. To burn a bootable SD card using a Linux machine, the user should install the release on a Linux machine, and then follow the instructions on the *Linux SD Card Creation Wiki*.

Code may be developed on the target, or on an external Linux machine using the cross compiler, and other tools. To enable cross compiler, the user must install ti-processor-SDK-am57xx on a Linux machine using [Processor-SDK-LINUX-AM57X 02_00_01_07](#), and the appropriate tools. The instructions on how to install the ti-processor-SDK-am57xx on a Linux machine are in the documentation on the same download page.

AM57xx Linux SDK Documentation	
Adding support for 64-bit Operating Systems	Instructions for users using a 64-bit version of Ubuntu
Processor SDK Linux Release Notes	Link to Release Notes for Processor SDK Linux
AM572x_EVM_QSG_7-1-15_beta.pdf	AM57xx EVM Quick Start Guide
processor-sdk-linux-gsg-01.00.01.00.pdf	Getting Started Guide
Wiki version of Software Developers Guide	Link to the online Software Developers Guide which has the latest content
Software Manifest	Software Manifest of Components Inside the SDK
AM57xx Linux SDK Checksums	

Figure 6. Processor-SDK-Linux AM57X Install Page Continued

In this TI design the code is built on the target using the internal code generation tools for the ARM Cortex-A15 processor and the C66x DSPs.

2.2 ***Install FTDI Drivers and Terminal Console Program***

This section will explain how to install the FTDI drivers and terminal console program.

1. Install a terminal console program such as Tera-Term, Putti, picocom, or minicom on a PC.
2. Install FTDI drivers from ftdichip.com.
3. Connect the FTDI-to-USB cable to the AM57EVM board where the green wire is the furthest away from the power jack, and the black wire is close to the power, as shown in [Figure 7](#).

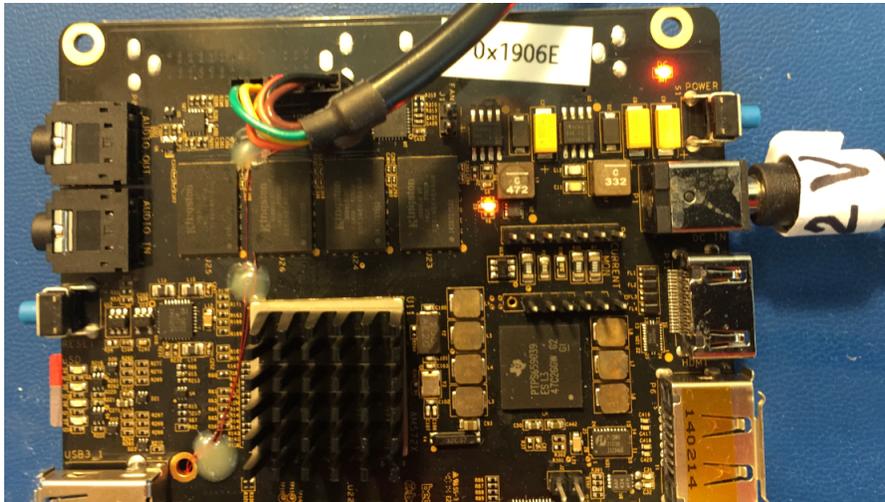


Figure 7. Connecting The FTDI Cable

To connect to the EVM, the terminal console parameters are as follows:

- Baud rate 115200
- 8-bit data
- No parity
- 1-bit stop
- No flow control

2.4 Build the Applications

The standard processor SDK release contains several OpenCL examples. They are in the `/usr/shared/ti/examples/` directory. Instructions how on to build and run the release examples are given in the [Processor SDK LINUX Software Developer's Guide](#).

This TI design has two programs. The first generates a sequence of 32-K Gaussian random variables using only the ARM cores and benchmarks the average time it takes to generate one Gaussian variable.

NOTE: In future releases, the Monte Carlo simulation will be part of the examples in the release. Thus, the example will be built with all other examples.

A second program is an OpenCL program. This program uses the DSP cores to generate sequences of Gaussian random variables; each sequence has 32-K values. When the DSP cores generate sequence N, the ARM cores process the previous sequence, sequence N-1. [Figure 10](#) shows the flow of the program.

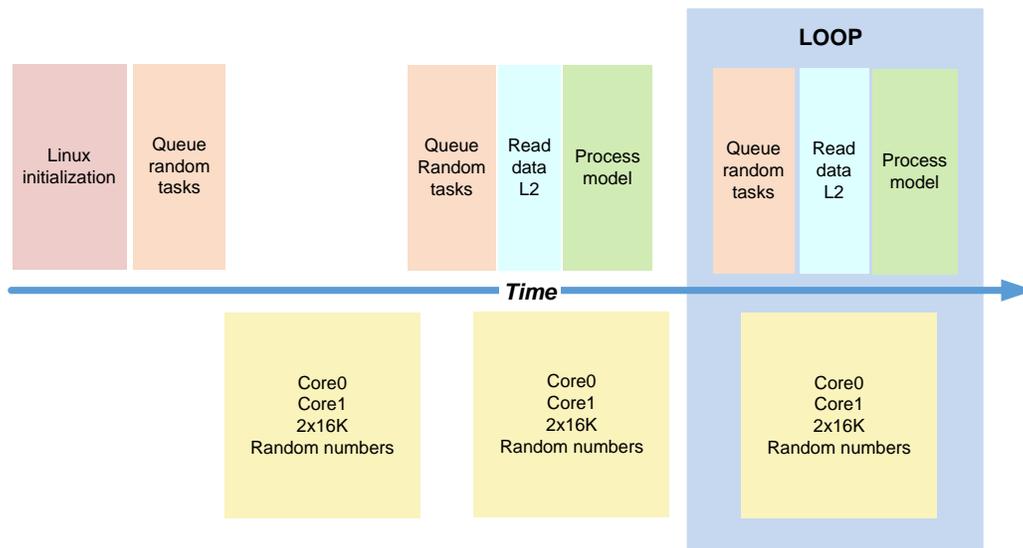


Figure 10. Program Flow

1. Create a new directory for the two projects.
2. Log in as root do **mkdir opencil**.
3. Change directory to the new directory **cd opencil**.
4. Copy the Makefile `cp /usr/shared/ti/examples/opencil/Makefile`.
5. Copy the make.inc file `cp /usr/shared/ti/examples/opencil/make.inc` ([Figure 11](#) shows what appears on the screen).

```

root@am57xx-evm:~#
root@am57xx-evm:~#
root@am57xx-evm:~# pwd
/home/root
root@am57xx-evm:~# mkdir opencil
root@am57xx-evm:~# cd opencil
root@am57xx-evm:~/opencil# cp /usr/share/ti/examples/opencil/Makefile .
root@am57xx-evm:~/opencil# cp /usr/share/ti/examples/opencil/make.inc .
root@am57xx-evm:~/opencil# ls -ltr
-rwxr-xr-x  1 root  root    548 Aug 26 13:45 Makefile
-rwxr-xr-x  1 root  root   2450 Aug 26 13:45 make.inc
root@am57xx-evm:~/opencil#
    
```

Figure 11. Screen Shot 2

6. Build the ARM-only Monte-Carlo Simulation.

NOTE: The source code for the ARM-only Monte-Carlo simulation is stored as a TAR file in the design sources of this TI design.

The following instructions use scp from a Ubuntu machine to the AM57 EVM.

7. Push the monte_carlo_arm simulation into the AM57 EVM using the scp utility.
8. Load the monte_carlo_arm.tar file into a PC.
9. Find the IP address of the AM57 EVM do ifconfig, as shown in [Figure 12](#)

NOTE: In [Figure 12](#), the IP address is 158.218.109.224.

10. Write down the system IP address.

```

root@am57xx-evm:~/openc1#
root@am57xx-evm:~/openc1#
root@am57xx-evm:~/openc1#
root@am57xx-evm:~/openc1# ifconfig
eth0      Link encap:Ethernet  HWaddr 7C:66:9D:EC:F2:BC
          inet addr:158.218.109.224  Bcast:0.0.0.0  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:10987 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2042756 (1.9 MiB)  TX bytes:684 (684.0 B)
          Interrupt:101

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:125 errors:0 dropped:0 overruns:0 frame:0
          TX packets:125 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:228986 (223.6 KiB)  TX bytes:228986 (223.6 KiB)

```

Figure 12. Screen Shot 3

11. Go to the directory in the Ubuntu machine where the monte_carlo_arm.tar file is.
12. Use the scp to copy the file into the openc1 directory.

NOTE: The file appears as scp monte_carlo_arm.tar.

If you are asked to update the secure addresses, follow the system prompt or agree to the transfer.

```

student1@ULA0270985: ~/compressed/openc1_09_21/openc1/old$
student1@ULA0270985: ~/compressed/openc1_09_21/openc1/old$ scp monte_carlo_arm.ta
r root@158.218.109.224:openc1/
monte_carlo_arm.tar                                100% 20KB 20.0KB/s 00:00
student1@ULA0270985: ~/compressed/openc1_09_21/openc1/old$

```

Figure 13. Screen Shot 4

13. Verify that the openc1 directory on the AM57 EVM has the tar file.
14. To un-tar the monte_carlo_arm.tar, type in tar -xvf monte_carlo_arm.tar as shown in [Figure 14](#).

```

root@am57xx-evm:~/openc1# ls -ltr
-rwxr-xr-x  1 root  root           548 Aug 26 13:45 Makefile
-rwxr-xr-x  1 root  root          2450 Aug 26 13:45 make.inc
-rw-r--r--  1 root  root         20480 Aug 26 14:25 monte_carlo_arm.tar
root@am57xx-evm:~/openc1#

```

Figure 14. Screen Shot 5

15. U-tar the monte_carlo_arm,tar file by typing `tar -xvf monte_carlo_arm.tar` as shown in [Figure 15](#).

```
EXE      = Monte_Carlo_ARM
CPP_FLAGS = -O3 -mcpu=neon -ftree-vectorize -funsafe-math-optimizations -mcpu=
CLOCL_FLAGS =

include ../make.inc

$(EXE): main.o
    @$(CPP) $(CPP_FLAGS) main.o $(LD_FLAGS) $(LIBS) -lrt -o $@

~
```

Figure 15. Screen Shot 6

16. Go to the monte_carlo_arm directory `cd`.
17. To view the contents of the monte_carlo_arm makefile, use the `vi`, `more`, or `cat` utilities.
18. Observe that the ARM code is compiled with optimization (`-O3`) and the A15 special properties (Neon™) are enabled as shown in [Figure 16](#).

```
EXE      = Monte_Carlo_ARM
CPP_FLAGS = -O3 -mcpu=neon -ftree-vectorize -funsafe-math-optimizations -mcpu=
CLOCL_FLAGS =

include ../make.inc

$(EXE): main.o
    @$(CPP) $(CPP_FLAGS) main.o $(LD_FLAGS) $(LIBS) -lrt -o $@

~
```

Figure 16. Screen Shot 7

19. Look at the source code and include the file.
20. See the algorithm to generate the Gaussian Random Sequence.
21. Build the monte_carlo_arm executable `cd` to the OpenCL directory `cd`.
22. Run `make`.
23. Run the monte_carlo_arm program by returning to the monte_carlo_arm directory `cd monte_carlo_arm`.
24. Verify the executable by typing in `./Monte_Carlo_ARM` as shown in [Figure 17](#).

```
root@am57xx-evm:~/opencl/monte_carlo_arm#
root@am57xx-evm:~/opencl/monte_carlo_arm# cd ..
root@am57xx-evm:~/opencl# make
===== monte_carlo_arm =====
Compiling main.cpp
root@am57xx-evm:~/opencl# cd monte_carlo_arm
root@am57xx-evm:~/opencl/monte_carlo_arm# ls -ltr
-rw-r--r-- 1 1001  tisdk  2895 Aug 17 21:34 utilityRoutines.h
-rw-r--r-- 1 1001  tisdk  3825 Aug 17 21:34 generateRandomGaussian.h
-rw-r--r-- 1 1001  tisdk   909 Aug 17 21:34 gaussRandom.h
-rw-r--r-- 1 1001  tisdk   309 Aug 17 21:34 Makefile
-rw-r--r-- 1 1001  tisdk  4178 Aug 26 14:50 main.cpp
-rw-r--r-- 1 root   root   3800 Aug 26 14:50 main.o
-rwxr-xr-x 1 root   root   8736 Aug 26 14:50 Monte_Carlo_ARM
root@am57xx-evm:~/opencl/monte_carlo_arm# ./Monte_Carlo_ARM
open Out file $i measuring Gaussian time 3615755.000000 nano secs
processed 32768.000000 elements , nanosecond per element 110.344086
DONE
```

Figure 17. Screen Shot 8

The accelerator version of the Monte-Carlo simulation uses the DSP to generate Gaussian Random numbers and uses the ARM processor to execute a function that applies the Gaussian Random numbers generated previously. The ARM function creates a simple histogram; the function counts the number of values greater than zero and the number of values that are less or equal to zero. The function prints the two counts on the screen.

25. Build the accelerator version of the Monte-Carlo simulation.
26. Go to the source code for the OpenCL Monte-Carlo simulation.
27. Select the file named monte_carlo_openc1.tar.
28. Push the file into the AM57 EVM.
29. To un-tar the monte_carlo_openc1.tar file, select tar-xvf monte_carlo_openc1 as shown in [Figure 18](#).

```

root@am57xx-evm:~/openc1#
root@am57xx-evm:~/openc1#
root@am57xx-evm:~/openc1# tar -xvf monte_carlo_openc1.tar
monte_carlo_openc1_new/
monte_carlo_openc1_new/cpu_main.cpp
monte_carlo_openc1_new/dsp_ccode.c
monte_carlo_openc1_new/Monte_Carlo_Opencl
monte_carlo_openc1_new/initial.h
monte_carlo_openc1_new/dsp_kernels.dsp_h
monte_carlo_openc1_new/Makefile
monte_carlo_openc1_new/MonteCarloSimulationExampleforOpenC_manifest.html
monte_carlo_openc1_new/dsp_kernels.cl
monte_carlo_openc1_new/show.py
root@am57xx-evm:~/openc1# █
  
```

Figure 18. Screen Shot 9

30. Go to the monte_carlo_openc1 directory cd titled monte_carlo_openc1_new.
31. Use vi, more, or cat to review the makefile.
32. Observe that the DSP flags are set with -O3, which indicates no symbolic debug.
33. Notice that the -k keeps the assembly file of the DSP code as shown in [Figure 19](#).

```

EXE      = Monte_Carlo_Opencl
CPP      = g++
CPP_FLAGS = -O3
CL6X     = cl6x -m6600 -k --abi=eabi -I$(TI_OCL_CGT_INSTALL)/include
CL6X_FLAGS = -O3 -I/usr/share/ti/openc1
  
```

Figure 19. Screen Shot 10

NOTE: View the `cpu_main.cpp` code and notice how the OpenCL code is built. The OpenCL wrapper, `dsp_kernel.cl`, is used to dispatch the DSP kernels. The file `dsp_ccode.c` with the included files are standard C code (without intrinsic). These files are used to generate the Gaussian Random sequence.

34. Build the monte_carlo_openc1 executable cd to the openc1 directory **executable**.
35. Run **make**.

NOTE: The first time that the code is built it will take a few minutes. [Figure 20](#) shows this process.

```

root@am57xx-evm:~/openc1/monte_carlo_openc1_new#
root@am57xx-evm:~/openc1/monte_carlo_openc1_new#
root@am57xx-evm:~/openc1/monte_carlo_openc1_new#
root@am57xx-evm:~/openc1/monte_carlo_openc1_new#
root@am57xx-evm:~/openc1/monte_carlo_openc1_new#
root@am57xx-evm:~/openc1/monte_carlo_openc1_new#
root@am57xx-evm:~/openc1/monte_carlo_openc1_new# make
compiling dsp_ccode.c for execution on DSP
compiling dsp_kernels.cl for execution on DSP
compiling cpu_main.cpp for execution on CPU
linking Monte_Carlo_Opencl for execution on CPU
  
```

Figure 20. Screen Shot 11

36. Run the monte_carlo_openc1 program by returning to the monte_carlo_openc1 directory **cd** titled **monte_carlo_openc1**.
37. Verify that the executable Monte_Carlo_Opencl was built by selecting **ls-ltr**.
38. Run the executable by selecting `./Monte_Carlo_Opencl` as shown in [Figure 21](#) and [Figure 22](#).

```

root@am57xx-evm:~/openc1/monte_carlo_openc1_new#
root@am57xx-evm:~/openc1/monte_carlo_openc1_new#
root@am57xx-evm:~/openc1/monte_carlo_openc1_new# ls -ltr
-rw-r--r-- 1 root root 13177 Aug 26 13:30 MonteCarloSimulationExampleforOpenC_manifest.html
-rwxr-xr-x 1 root root 161 Oct 5 23:22 show.py
-rw-r--r-- 1 root root 3539 Oct 5 23:22 dsp_kernels.cl
-rw-r--r-- 1 root root 18202 Oct 5 23:22 dsp_ccode.c
-rw-r--r-- 1 root root 8743 Oct 5 23:22 cpu_main.cpp
-rw-r--r-- 1 root root 4242 Oct 6 11:56 initial.h
-rw-r--r-- 1 root root 1922 Oct 7 09:00 Makefile
-rw-r--r-- 1 root root 82677 Oct 7 11:24 dsp_kernels.dsp.h
-rwxr-xr-x 1 root root 43417 Oct 7 11:24 Monte_Carlo_Opencl
root@am57xx-evm:~/openc1/monte_carlo_openc1_new#
    
```

Figure 21. Screen Shot 12

```

root@am57xx-evm:~/openc1/monte_carlo_openc1_new#
root@am57xx-evm:~/openc1/monte_carlo_openc1_new#
root@am57xx-evm:~/openc1/monte_carlo_openc1_new# ./Monte_Carlo_Opencl
[ 5284.366414] NET: Registered protocol family 41
*****
Total Time (ns): 880000 per value (ns): 26
number of elements: 32768
count positive = 16293 count Negative = 16475
*****
Total Time (ns): 700000 per value (ns): 21
number of elements: 32768
count positive = 16396 count Negative = 16372
*****
Total Time (ns): 682000 per value (ns): 20
number of elements: 32768
count positive = 16342 count Negative = 16426
*****
Total Time (ns): 657000 per value (ns): 20
number of elements: 32768
count positive = 16389 count Negative = 16379
*****
Total Time (ns): 648000 per value (ns): 19
number of elements: 32768
count positive = 16326 count Negative = 16442
*****
Total Time (ns): 647000 per value (ns): 19
number of elements: 32768
count positive = 16385 count Negative = 16383
*****
Total Time (ns): 647000 per value (ns): 19
number of elements: 32768
count positive = 16335 count Negative = 16433
*****
Total Time (ns): 646000 per value (ns): 19
number of elements: 32768
count positive = 16388 count Negative = 16380
*****
Total Time (ns): 644000 per value (ns): 19
number of elements: 32768
count positive = 16333 count Negative = 16435
*****
END RUN!
root@am57xx-evm:~/openc1/monte_carlo_openc1_new#
    
```

Figure 22. Screen Shot 13

3 Benchmarks

When the ARM only code runs, the code takes about 110 nanoseconds to generate a single Gaussian Random number. When the OpenCL is used to take advantage of the DSP cores, generating a single Gaussian Random number takes 19 to 20 nanoseconds. During the generation time, the ARM processor executes a different function that inputs the generated Gaussian sequence.

4 References

1. *Generating Gaussian Random Numbers*, <http://www.design.caltech.edu/erik/Misc/Gaussian.html>
2. *Random Number Generator Algorithm*, http://www.cryptosys.net/rng_algorithms.html#topofpage
3. *Random-Number Generation*, http://www.mi.fu-berlin.de/inf/groups/ag-tech/teaching/2012_SS/L_19540_Modeling_and_Performance_Analysis_with_Simulation/06.pdf
4. *Parallel Pseudorandom Number Generation*, <https://www.siam.org/pdf/news/744.pdf>
5. [SDK-AM57x Processor](#)
6. *AM572x Evaluation Module Quick Start Guide*,
7. *Khronos Conformat Products*, <https://www.khronos.org/conformance/adopters/conformant-products>

5 About the Author

RAN KATZUR is a senior application engineer at TI where he supports the Sitara and the DSP families of System-on-a-Chip (SOC) devices. Ran brings to this role his extensive experiences and knowledge in parallel processing and optimization. Ran earned B.Sc., M.Sc. and Ph.D. in applied mathematics from Tel-Aviv University.

IMPORTANT NOTICE FOR TI REFERENCE DESIGNS

Texas Instruments Incorporated ("TI") reference designs are solely intended to assist designers ("Buyers") who are developing systems that incorporate TI semiconductor products (also referred to herein as "components"). Buyer understands and agrees that Buyer remains responsible for using its independent analysis, evaluation and judgment in designing Buyer's systems and products.

TI reference designs have been created using standard laboratory conditions and engineering practices. **TI has not conducted any testing other than that specifically described in the published documentation for a particular reference design.** TI may make corrections, enhancements, improvements and other changes to its reference designs.

Buyers are authorized to use TI reference designs with the TI component(s) identified in each particular reference design and to modify the reference design in the development of their end products. HOWEVER, NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY THIRD PARTY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT, IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI REFERENCE DESIGNS ARE PROVIDED "AS IS". TI MAKES NO WARRANTIES OR REPRESENTATIONS WITH REGARD TO THE REFERENCE DESIGNS OR USE OF THE REFERENCE DESIGNS, EXPRESS, IMPLIED OR STATUTORY, INCLUDING ACCURACY OR COMPLETENESS. TI DISCLAIMS ANY WARRANTY OF TITLE AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, QUIET ENJOYMENT, QUIET POSSESSION, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS WITH REGARD TO TI REFERENCE DESIGNS OR USE THEREOF. TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY BUYERS AGAINST ANY THIRD PARTY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON A COMBINATION OF COMPONENTS PROVIDED IN A TI REFERENCE DESIGN. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, SPECIAL, INCIDENTAL, CONSEQUENTIAL OR INDIRECT DAMAGES, HOWEVER CAUSED, ON ANY THEORY OF LIABILITY AND WHETHER OR NOT TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, ARISING IN ANY WAY OUT OF TI REFERENCE DESIGNS OR BUYER'S USE OF TI REFERENCE DESIGNS.

TI reserves the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques for TI components are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

Reproduction of significant portions of TI information in TI data books, data sheets or reference designs is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards that anticipate dangerous failures, monitor failures and their consequences, lessen the likelihood of dangerous failures and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in Buyer's safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed an agreement specifically governing such use.

Only those TI components that TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components that have **not** been so designated is solely at Buyer's risk, and Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.