Redpaper

Dino Quintero
Wei Li
Wainer dos Santos Moschetta
Mauricio Faria de Oliveira
Alexander Pozdneev

NVIDIA CUDA on IBM POWER8: Technical overview, software installation, and application development

Overview

The exploitation of general-purpose computing on graphics processing units (GPUs) and modern multi-core processors in a single heterogeneous parallel system has proven highly efficient for running several technical computing workloads. This applied to a wide range of areas such as chemistry, bioinformatics, molecular biology, engineering, and big data analytics.

Recently launched, the IBM® Power System S824L comes into play to explore the use of the NVIDIA Tesla K40 GPU, combined with the latest IBM POWER8™ processor, providing a unique technology platform for high performance computing.

This IBM Redpaper™ publication discusses the installation of the system, and the development of C/C++ and Java applications using the NVIDIA CUDA platform for IBM POWER8.

Note: CUDA stands for *Compute Unified Device Architecture*. It is a parallel computing platform and programming model created by NVIDIA and implemented by the GPUs that they produce.

The following topics are covered:

- Advantages of NVIDIA on POWER8
- The IBM Power Systems S824L server
- Software stack
- System monitoring
- Application development
- Tuning and debugging
- Application examples

Advantages of NVIDIA on POWER8

The IBM and NVIDIA partnership was announced in November 2013, for the purpose of integrating IBM POWER®-based systems with NVIDIA GPUs, and enablement of GPU-accelerated applications and workloads. The goal is to deliver higher performance and better energy efficiency to companies and data centers.

This collaboration produced its initial results in 2014 with:

- The announcement of the first IBM POWER8 system featuring NVIDIA Tesla GPUs (IBM Power Systems™ S824L).
- The release of CUDA 5.5 for POWER8.
- The availability of several applications and tools to leverage GPU acceleration and CUDA (for example, IBM XL C/C++ compilers, IBM Java, and others).

More applications, middlewares, and workloads from various fields announced upcoming support for GPU acceleration on IBM Power Systems.

The computational capability provided by the combination of NVIDIA Tesla GPUs and IBM POWER8 systems enable workloads from scientific, technical, and high performance computing to run on data center hardware. In most cases, these workloads were run on supercomputing hardware. This computational capability is built on top of massively parallel and multithreaded cores with NVIDIA Tesla GPUs and IBM POWER8 processors, where processor-intensive operations were offloaded to GPUs and coupled with the system's high memory-hierarchy bandwidth and I/O throughput.

An overview of the NVIDIA Tesla GPU is provided in "NVIDIA Tesla K40 GPU" on page 10. You can learn more at the following website:

http://nvidia.com/tesla

For more information about the IBM POWER8 processor and systems, refer to "The IBM Power Systems S824L server" on page 2.

Moreover, the development and portability of GPU-accelerated applications for IBM Power Systems with NVIDIA GPUs is made easier with the availability of CUDA and little-endian mode on POWER8, which increases the commonality with other popular architectures, and the growing ecosystem built around the OpenPOWER Foundation. You can learn more about OpenPOWER at the following website:

http://openpowerfoundation.org

In summary, IBM Power Systems with NVIDIA GPUs provides a computational powerhouse for running applications and workloads from several scientific domains, and for processing massive amounts of data, with eased application development and portability. All of this builds upon the strong ecosystems of the NVIDIA CUDA architecture and the OpenPOWER Foundation.

The IBM Power Systems S824L server

In October 2014, IBM announced a new range of systems targeted at handling massive amounts of computational data[1].

[1] See *IBM Provides Clients Superior Alternative to x86-Based Commodity Servers*, found at: http://www.ibm.com/press/us/en/pressrelease/45006.wss

IBM Power Systems S824L[2] (see Figure 1) is the first IBM Power Systems server that features NVIDIA GPU. This offering delivers a new class of technology that maximizes performance and efficiency for scientific, engineering, Java, big data analytics, and other technical computing workloads. Designed to empower the ecosystem of open source development, these new servers support little-endian Ubuntu Linux running in bare metal mode.

Figure 1 Front view of the IBM Power Systems S824L server

The IBM POWER8 processor was designed for data-intensive technical computing workloads, big data and analytics applications. All aspects of its design are optimized to deal with today's exploding data sizes. The POWER8 cache hierarchy was architected to provide data bandwidth that is suitable for running large data sets through and to accommodate their large footprints.

The POWER8 processor comes in two versions. One version is targeted toward large SMP systems at the enterprise level. The other is specifically designed for scale-out servers that are building blocks for current and future IBM high performance computing systems. The two latter chips are installed in pairs in a dual-chip module (DCM). DCM connects its chips with SMP links and plugs into a socket in the planar of the system. Functionally, DCM works as a single processor, and the operating system considers DCM as a NUMA node. The IBM POWER8 scale-out servers contain one or two sockets populated with DCMs.

The IBM Power Systems S824L server (model 8247-42L) is a two-socket system. The server is available in two-processor configurations:

- Two 10-core 3.42 GHz POWER8 DCMs (20 cores per system)
- Two 12-core 3.02 GHz POWER8 DCMs (24 cores per system)

The IBM Power Systems S824L server has 16 slots for DDR3 ECC memory[3] (8 slots per DCM) and supports memory modules of 16 GB, 32 GB, and 64 GB. The maximum configurable system memory is 1 TB. The form factor for the memory modules is Custom DIMM (CDIMM).

[2] The model name follows a simple convention. The first letter ("S") stands for "scale-out", the first digit indicates POWER8 processor, the second digit is for the number of sockets, the third digit reflects the server height in standard rack units, and the trailing "L" letter indicates "scale-out/Linux". IBM Power scale-out/Linux servers are designed specifically to run the Linux operating system. If you are interested in running the IBM AIX® or IBM i operating system on POWER scale-out servers, you need to consider server models that do not have the letter "L" suffix in their names (S814, S822, and S824: as of writing time of this IBM Redpaper publication).

[3] Error-correcting code (ECC) memory is a type of random access memory that can detect and correct spontaneous runtime failures of memory cells or memory access logic.

Note: In general, the introduction of new technology in memory modules may increase the maximum system memory configurable for a system. However, that is subject to support, compatibility, and possibly other restrictions.

The minimum system configuration includes one NVIDIA GPU. The maximum system configuration is two GPUs.

The S824L planar features 11 PCIe Gen3 slots:

- Four ×16 slots (one or two are used for NVIDIA GPU cards)
- Seven ×8 slots (one is used for the integrated LAN adapter)

The server has a rack-mounted form factor and takes 4U (four units) of rack space. It is depicted on Figure 1 on page 3. The lower-left frontal part of the server features 12 small form factor bays for internal storage based on HDDs (hard disk drives) or SSDs (solid-state drives). Two USB 3.0 ports and a DVD bay are visible at the lower-right front part of the server.

For more detailed information about the server options, search the IBM Offering Information website for the IBM Power Systems S824L:

http://www.ibm.com/common/ssi/index.wss

The official technical documentation for the IBM Power Systems S824L server is at:

http://www.ibm.com/support/knowledgecenter/8247-42L/p8hdx/8247_42l_landing.htm

To acquire a better technical understanding of the IBM Power Systems S824L server and its reliability, availability, and serviceability features, refer to the following IBM Redbooks publication: *IBM Power Systems S824L Technical Overview and Introduction*, REDP-5139.

The IBM POWER8 processor chip

The POWER8 processor is fabricated with IBM 22 nanometer (22 nm) silicon-on-insulator (SOI) technology using copper interconnects and 15 layers of metal. The scale-out version of the POWER8 chip shown in Figure 2 on page 5 contains six cores[4] and is 362 square millimeters (mm) in size. Each core has 512 KB of static RAM (SRAM) second-level cache (L2), and 8 MB of embedded DRAM (eDRAM) third-level cache (L3). The L3 cache is shared among all cores of the chip. The chip also includes a memory controller with four memory interfaces, PCIe Gen3 controllers, SMP interconnect, hardware accelerators, and coherent accelerator processor interface (CAPI). The interconnection system links all components within the chip.

[4] The number of cores activated depends on a server offering.

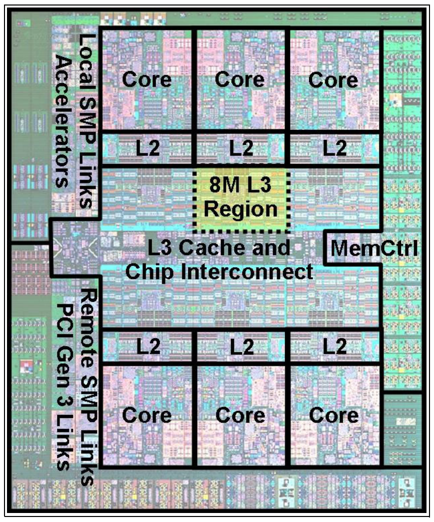


Figure 2 The POWER8 scale-out processor chip

The SMP fabric of the IBM POWER8 scale-out chip has two types of links:

- "X" bus connects a pair of chips to form a dual-chip module (DCM)
- "A" bus provides intersocket communications

Two POWER8 scale-out chips are installed in pairs in a DCM that plugs into a socket in a system planar of a computing server. In a maximum configuration, 12 POWER8 cores of a DCM share 96 MB of L3 cache and two memory controllers address 512 GB of system memory through eight memory interfaces.

POWER8 core

The POWER8 is a 64-bit processor based on the IBM Power (Performance Optimization With Enhanced RISC[5]) Instruction Set Architecture (ISA) Version 2.07. The POWER8 computing core is a superscalar out-of-order eight-way simultaneously multithreaded (SMT)[6] microprocessor. POWER8 is the first processor based on the IBM POWER Architecture that supports little-endian as well as big-endian byte ordering in virtualized and non-virtualized (or bare-metal) mode[7].

The primary components of a POWER8 core are shown in Figure 3 on page 6:

- Instruction fetch unit (IFU)
- Instruction sequencing unit (ISU)
- Load/store unit (LSU)
- Fixed point unit (FXU)
- Vector and scalar unit (VSU)
- Decimal floating point unit (DFU)

[5] RISC is an acronym for "reduced instruction set computer". The first prototype computer to use RISC architecture was designed by IBM researcher John Cocke and his team in the late 1970s. For the historical perspective, see http://www.ibm.com/ibm/history/ibm100/us/en/icons/risc

[6] Simultaneous multithreading capabilities of POWER8 processor core allow eight independent software threads to efficiently share the core resources. This mode is known as SMT8. In a single-threaded mode, almost all the resources of the highly parallel POWER8 core are used by the single thread.

[7] The byte ordering (big-endian or little-endian) for a storage access is specified by the operating system.

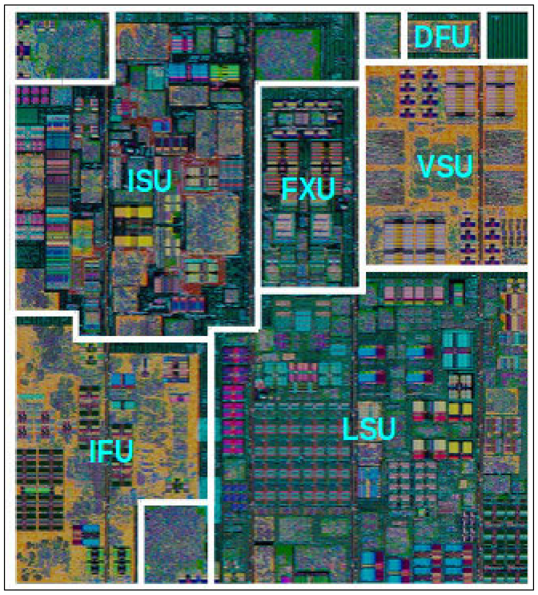


Figure 3 The IBM POWER8 processor core

On each cycle, the IFU of a POWER8 core selects a thread and fetches eight quadword-aligned[8] instructions from the 32 KB eight-way first-level instruction cache into a thread instruction buffer. POWER8 instructions are always 4 bytes long and word-aligned. As a result, an eight-instruction block is 32-byte long.

In SMT modes, the POWER8 thread priority logic selects two threads per cycle for instruction group formation. Groups are independently formed by reading a maximum of three nonbranches and one branch from the instruction buffer of each thread, for a total of eight instructions. After group formation, the simple instructions are immediately decoded, and the complex instructions are routed to special microcode hardware that breaks them into a series of simple internal operations.

The POWER8 core dispatches instructions on a group basis as soon as all resources are available for the instructions in a group. In SMT modes, two groups are dispatched independently. The ISU dispatches eight instructions per cycle into three separate issue queues:

- A unified queue
- A branch issue queue
- A condition register issue queue

The ISU is also responsible for register renaming and instruction completion.

POWER8 core saves dispatched instructions in the issued queues and then issues them to the execution units. Instructions can be issued in order or out of order from all of these queues. An instruction in the issue queue is selected for issuing when all source operands for that instruction are available. The issue queues together can issue a total of ten instructions per cycle:

- One branch instruction to the branch execution unit
- One condition register logical instruction to the condition register execution unit
- Two fixed-point instructions to the FXU
- Two load/store and two load instructions to the LSU
- Two instructions to the VSU

[8] Halfword, word, doubleword, and quadword are 2-, 4-, 8-, and 16-byte memory entries, respectively. One, two, three, and four least significant bits of a halfword-, word-, doubleword-, and quadword-aligned memory address are zeros, accordingly.

In addition to load/store operations, the LSU can accept and execute simple fixed-point instructions coming from each of its four execution pipelines.

VSU includes the following subunits:

- Two single precision vector floating point units (FPUs)
- Two double precision vector FPUs
- Two scalar FPUs
- Two fixed-point vector units
- One decimal floating point unit (DFU)
- One cryptographical operations unit

VSU vector subunits operate on 128-bit registers. To manipulate vectors, VSU implements vector multimedia extension (VMX) and vector scalar extension (VSX) instructions[9]. The POWER8 core is able to issue up to two fused multiply-add instructions on vectors in each cycle, thereby delivering 16 single precision or eight double precision floating point operations per cycle during peak workloads.

PCI Express

The IBM POWER8 server system components are connected through the Peripheral Component Interconnect Express Gen3 (PCI Express Gen3 or PCIe Gen3) bus.

In server systems, some PCIe devices are connected directly to the PCIe Gen3 buses on the processors, and other devices are connected to these buses through PCIe Gen3 switches. PCIe slots are used to plug in devices such as InfiniBand cards, 10 GbE network interface cards, Fibre Channel adapters, SAS cards for internal disks and external SAS ports. Some of the PCIe slots are Coherent Accelerator Processor Interface (CAPI)-enabled to connect custom acceleration engines (for example, (field-programmable gate arrays (FPGAs)) to the coherent fabric of the POWER8 chip.

The IBM POWER8 scale-out chip has 24 PCIe Gen3 full duplex lanes that provide 7.877 Gbit/s of effective bandwidth in each direction. The total effective theoretical bandwidth for a DCM is calculated as follows:

```
2 chips × 24 lanes × 7.877 Gbit/s × 2 directions = 94.524 Gbyte/s
```

Note: The bandwidth listed here might appear slightly differently from other materials. In our case, we consider the bandwidth overhead originated from the PCIe Gen3 encoding scheme. We also avoid rounding where possible.

Energy awareness

The energy that is required to power and cool servers contributes significantly to the overall operational efficiency of the computing infrastructure. To address this challenge, IBM developed the EnergyScale™ technology for IBM Power Systems servers. This technology helps to control the power consumption and performance of POWER8 servers. For more details, refer to the IBM EnergyScale for POWER8 processor-based systems paper at:

http://public.dhe.ibm.com/common/ssi/ecm/en/pow03125usen/POW03125USEN.PDF

The IBM POWER8 chip has an embedded IBM PowerPC® 405 processor with 512 KB of SRAM. This processor known as an on-chip controller (OCC) runs a real-time control firmware. The purpose of the OCC is to respond timely to workload variations. The OCC adjusts the per-core frequency and voltage based on activity, thermal, voltage, and current sensors. The OCC real-time OS was released as open source software.

[9] The IBM VMX and VSX instructions implement and extend the AltiVec specifications that were introduced to the Power ISA at its 2.03 revision.

See the open-power/OCC site:

http://github.com/open-power/occ

On-chip accelerators

The IBM POWER8 processor has the on-chip accelerators that provide the following functions:

- On-chip encryption
- On-chip compression
- On-chip random number generation

The Linux kernel uses on-chip accelerators through specialized drivers[10].

The on-chip cryptography accelerator (also known as the *cryptographical module*) provides high-speed encryption capabilities. The operating system can employ this facility to offload certain latency tolerant operations, for example, file system encryption and securing Internet Protocol communications.

The on-chip compression engine (sometimes referred to as *memory compression module*) was designed as a facility that helps the operating system to compress the least frequently used virtual memory pages.

The on-chip random number generator (typically abbreviated as *RNG*) provides the operating system with the source of hardware-based random numbers. This generator was architected to be cryptographically stronger than software-based pseudo-random number generators. In some instances, there can also be a performance advantage.

In addition to on-chip accelerators, each core has built-in functions that facilitate the implementation of cryptographical algorithms. An application developer can leverage these capabilities by employing compiler intrinsics in their source code or writing in assembly language.

Coherent accelerator processor interface

Many computing workloads can benefit from running on a specialized hardware like field-programmable gate arrays (FPGAs) or GPUs rather than on a general-purpose processor. Traditionally, external accelerators communicate with the processor through the I/O bus, which follows the model of I/O device operation. To simplify the interaction between processor and accelerators, IBM introduced the CAPI with the POWER8 processor. CAPI attaches devices to the SMP fabric of the POWER8 chip. As a result, CAPI enables off-chip accelerators to access the main system memory and participate in the system memory coherence protocol as a peer of other caches in the system. In other words, CAPI allows a specialized accelerator to be seen as just an additional processor in the system.

The conforming accelerator is to be plugged into a standard PCIe Gen3 slot marked as CAPI-enabled. The CAPI coherency protocol is tunneled over a standard PCIe Gen3 bus.

Memory subsystem

The IBM POWER8 processor was designed to have a strong memory subsystem to be capable of meeting the bandwidth, latency, and capacity demands of data and compute intensive applications.

[10] Available only in selected offerings.

Each core of the POWER8 processor has private caches of first (L1) and second level (L2):

- 32 KB of L1 instruction cache
- 64 KB of L1 data cache
- 512 KB of L2 unified cache

Cache memory of a third level (L3) is shared between all cores of the POWER8 chip. A core has faster access to its local 8 MB L3 cache region.

The scale-out version of the POWER8 processor chip has one memory controller with four memory interfaces. Memory slots are populated with custom DIMM (CDIMM) memory modules. *CDIMM* is a memory card that houses a set of industry standard dynamic random access memory (DRAM) memory chips and a memory buffer chip. The memory controller of a processor accesses system memory through that external memory buffer chip. The memory buffer chip isolates processor memory controller from the interaction with the actual DRAM memory chips. At the time this publication was written, the POWER8 server offerings include DDR3 memory options, whereas the described technology allows easy transition to DDR4 memory later. Figure 4 illustrates the POWER8 memory organization.

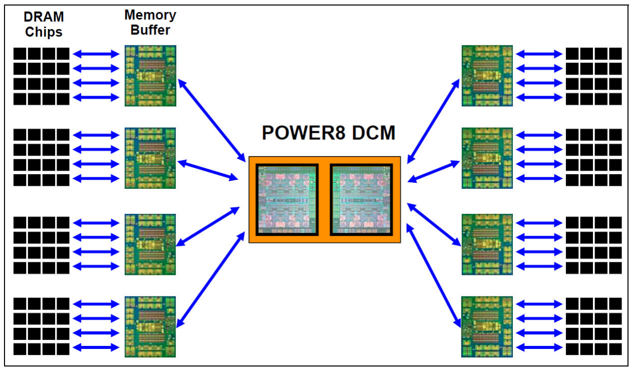


Figure 4   POWER8 memory organization

The memory buffer chip shown in Figure 5 consists of the following components:

- POWER8 processor link
- Memory cache
- Memory scheduler
- Memory manager
- DDR3 interfaces

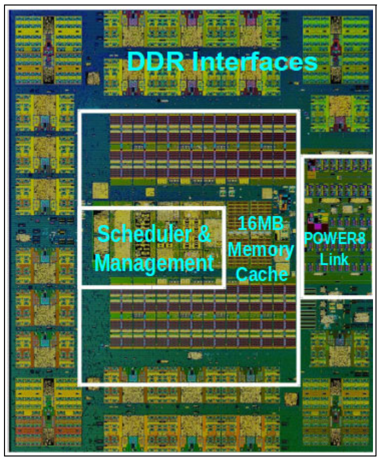


Figure 5   Memory buffer chip

The 16 MB memory buffer chip cache constitutes the fourth level (L4) of cache hierarchy. The memory scheduler supports prefetch and write optimization. The memory manager is responsible for RAS decisions[11] and energy management.

Hardware transactional memory

The *transactional memory* (TM) is a computer science concept that was created to facilitate the design of multithreaded algorithms. The TM makes a series of loads and stores appear as a single atomic operation, a *transaction,* that either succeeds or fails. The TM is primarily intended to be used in situations where software engineers typically employ locks in order to ensure atomicity of a set of operations.

There are algorithms in which the probability of a memory conflict between application threads is negligible. Nevertheless, even in such cases, software engineers need to employ locks in order to ensure that the algorithm is always correct. The use of locks in frequently executed pieces of code might have a dramatic impact on performance.

One algorithmic approach that allows you to avoid acquiring locks in this situation is a lock elision technique. A thread needs to mark the beginning and end of a *transaction* on shared variables with the TM directives or instructions provided by a programming language or system software. In an optimistic execution scenario, there are no conflicts, and transactions always succeed. In an expected highly improbable conflict situation, the TM reports that a transaction has failed. In this case, an algorithm developer can either try a transaction again with a TM, or fall back to a lock-based if-then-else branch in the code.

Historically, the TM was first implemented as a pure software-based mechanism. Hardware support for the TM became a commercial reality only recently. To the best of our knowledge, the first commercially available product with the hardware transactional memory (HTM) capabilities was the IBM Blue Gene/Q® (BG/Q) supercomputer in 2011. The BG/Q compute chip was based on PowerPC A2 processor cores and implemented transactional execution primarily in the L2 cache, which served as the point of coherence. In 2012, IBM started to ship IBM zEnterprise® EC12 (zEC12) IBM System z® mainframe solutions that targeted business audiences. With the zEC12 microprocessor, IBM offered a multiprocessor transactional support. At the time this publication was written, the latest IBM product with support for HTM is the POWER8 processor. The first POWER8 based server systems were shipped in 2014. The IBM POWER architecture support for HTM includes instructions and registers that control the transactional state of a thread. The Power ISA Version 2.07 specification also describes the behavior of transactions regarding the execution and memory models of the processor. It also includes more complex behaviors such as suspend mode, which allows non-transactional operations inside a transaction.

NVIDIA Tesla K40 GPU

The NVIDIA Tesla K40 GPU is based on Kepler architecture version GK110B that supports the CUDA compute capability 3.5. It can deliver 1.43 Tflop/s (peak) performance for double-precision floating point operations and 4.29 Tflop/s for single-precision.

The Kepler GK110B architecture highlights the following improvements over previous generations:

- Streaming Multiprocessor (SMX): Features a new generation of Streaming Multiprocessor (SM)
- Dynamic parallelism: Ability to launch nested CUDA kernels
- Hyper-Q: Allows several CPU threads or processes to dispatch CUDA kernels concurrently

[11] The RAS decisions deal with reliability, availability, and serviceability features and functions.

As a result, the NVIDIA Tesla K40 GPU can be logically viewed as follows:

- Fifteen SMX multiprocessors, each with:
  - 192 single-precision CUDA cores
  - 64 double-precision units
  - 32 special function units (SFU)
  - 32 load/store units
  - 16 texture filtering units
- Four warp schedulers and eight instruction dispatch units per SMX
- Six 64-bit memory controllers per GPU

Each thread can address up to 255 registers. The single and double-precision arithmetic units are fully compliant with IEEE 754-2008 standard, including an implementation of fused multiply-add (FMA).

A warp is a group of 32 parallel threads that are scheduled to run in an SMX multiprocessor. The maximum number of warps per SMX is 64, and it is capable of issuing and executing four warps simultaneously. Figure 6 shows the organization between multiprocessors, warps, and threads with the Tesla K40 GPU.

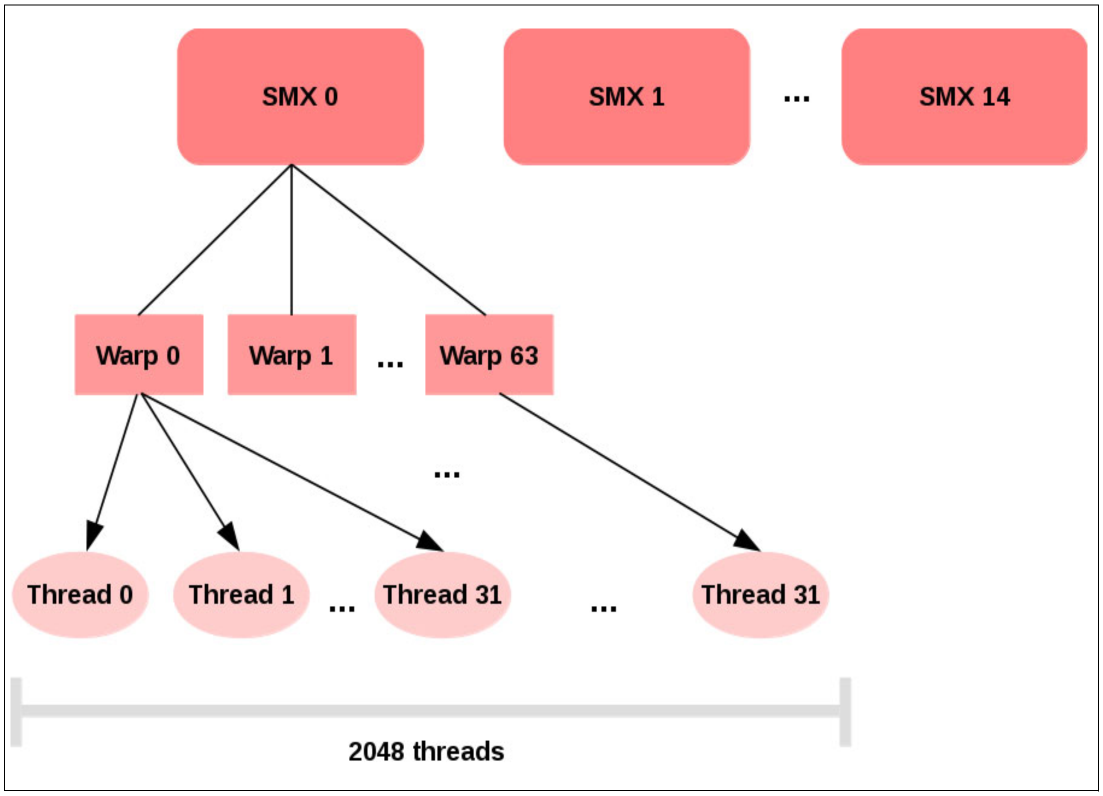


Figure 6 NVIDIA Tesla K40 GPU warps organization

The memory hierarchy is shown in Figure 7 on page 12 and has the following levels:

- 64 KB configurable shared memory and L1 cache per multiprocessor.

  There are three possible configurations:

  - 48 KB shared memory and 16 KB L1 cache
  - 16 KB shared memory and 48 KB L1 cache
  - 32 KB shared memory and 32 KB L1 cache

  The shared memory is accessed in words of 8 bytes.

- 48 KB read-only data cache per multiprocessor
- 1536 KB L2 cache
- 12 GB DRAM (GDDR5)

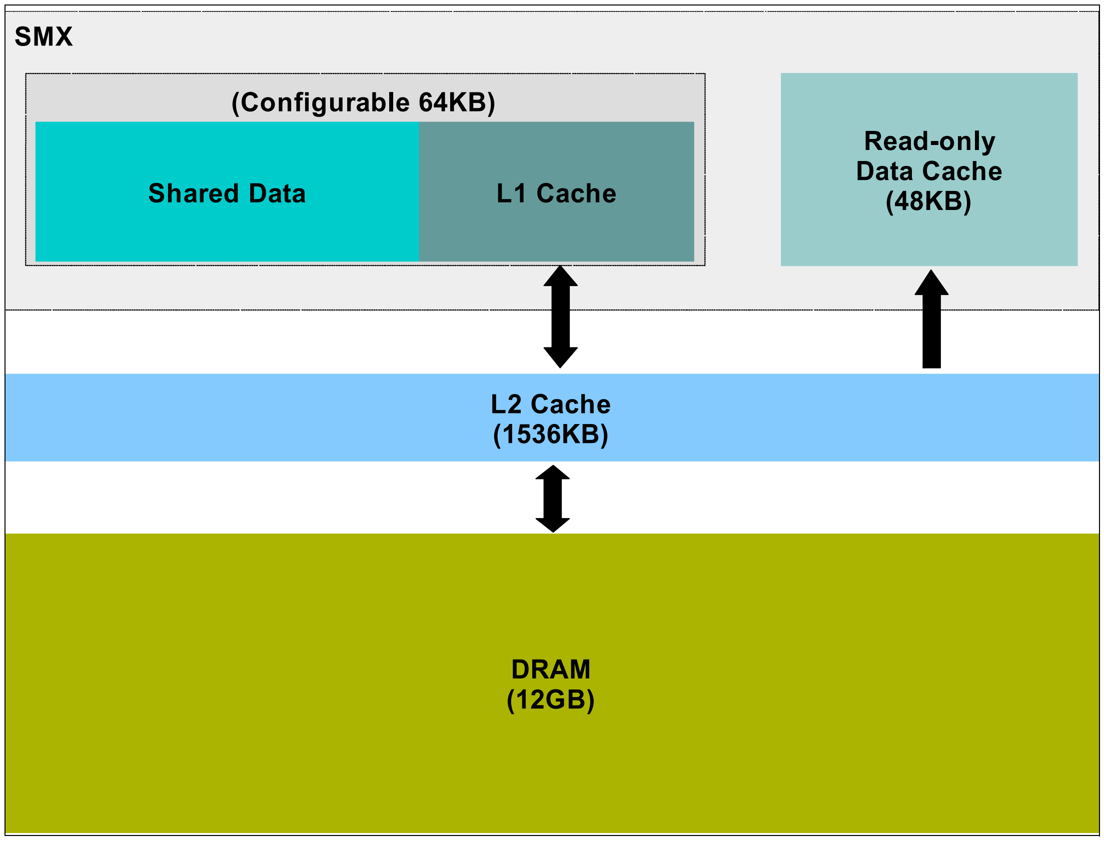


Figure 7 NVIDIA Tesla K40 GPU memory hierarchy

Memory errors in the shared memory, L1/L2 cache, and DRAM are remedied with the use of an error correcting code (ECC) implementation of a single-error correct or double-error detect (SECDED) scheme. The read-only data cache uses parity checking to detect problems and a single-error correction mechanism, which consists of reload data from the L2 cache.

The Hyper-Q technology (also known as *CUDA Streams*) is controlled by hardware that offers up to 32 connections to the work queues.

The NVIDIA GPU Boost feature is designed to speed up applications by taking advantage of the power headroom on the graphical card. A user application may not take full advantage of a 235 W Tesla K40 power budget. The NVIDIA GPU Boost feature allows users to change the default SMX clock frequency to higher values, while maintaining power consumption below the allowed limit. The processor clock options vary from 324 MHz up to 875 MHz with 745 MHz being the default value. The memory clock frequency is mostly kept at 3 GHz in any profile, except for the lowest 324 MHz processor clock frequency option as shown in Table 1 on page 13. Because 745 MHz is the base processor clock frequency, the 810 MHz and 857 MHz processor clock frequency options are used to boost performance.

Table 1 Tesla K40 GPU supported clocks

Memory clock (MHz)	**Processor clock (MHz)**
3004	875
3004	810
3004	745
3004	666
324	324

Detailed rationale and information about NVIDIA Tesla K40 Boost can be found in the *NVIDIA GPU Boost for Tesla K40 Passive and Active Boards - Application Note* found at this site:

http://www.nvidia.com/object/tesla_product_literature.html

One ×16 PCI Express Gen3 slot is used to connect a GPU card to the host system.

The Tesla K40 GPU has four compute modes:

- Prohibited: Not available for compute applications
- Exclusive Thread: Only one process and thread can use the GPU at a time
- Exclusive Process: Only one process can use the GPU at the time, but its threads can create work concurrently
- Default: Multiple processes/threads can use the GPU simultaneously

Example 1 shows the output of the *deviceQuery* application that comes with the CUDA toolkit samples (see “NVIDIA CUDA Toolkit code samples” on page 55). The command provides more information about the NVIDIA Tesla K40 GPU of the IBM Power Systems S824L server, which is discussed throughout this paper.

Example 1 Output of the deviceQuery application (some output omitted)

```
$ ./deviceQuery
<...>
Detected 2 CUDA Capable device(s)

Device 0: "Tesla K40m"
  CUDA Driver Version / Runtime Version          6.5 / 5.5
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 11520 MBytes (12079136768 bytes)
  (15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores
  GPU Clock rate:                                745 MHz (0.75 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096,
                                                 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   2 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

Device 1: "Tesla K40m"
<... output ommitted ...>
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
> Peer access from Tesla K40m (GPU0) -> Tesla K40m (GPU1) : No
> Peer access from Tesla K40m (GPU1) -> Tesla K40m (GPU0) : No

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.5, CUDA Runtime Version = 5.5,
NumDevs = 2, Device0 = Tesla K40m, Device1 = Tesla K40m
Result = PASS
```

Software stack

This section describes the installation of the software stack for running CUDA applications on POWER8 with the following components:

- CUDA: NVIDIA CUDA Toolkit v5.5 (CUDA 5.5) for POWER8
- Operating system: Ubuntu Server 14.10 in non-virtualized mode
- C/C++ compilers: GCC[12] 4.9 or IBM XL C/C++ compilers v13.1.1
- Java: IBM SDK, Java Technology Edition 7.1-2.0[13]

Note: NVIDIA announced an upcoming release of CUDA 7.0 as of this writing.

You should expect similarity with the content described in this paper, with changes where appropriate (for example, version numbers in command lines, and program output).

The supported operating systems for CUDA 7.0 on POWER8 now include Ubuntu Server 14.04.2 LTS, in addition to Ubuntu Server 14.10 (both in non-virtualized mode).

It also describes the installation of the cross-platform development packages for running a development environment on an x86-64 computer with CUDA applications running on the POWER8 system (remotely), with the following components:

- CUDA: NVIDIA CUDA Toolkit v5.5 (CUDA 5.5) for POWER8 (cross-platform on x86-64)
- IDE: NVIDIA NSight Eclipse Edition version 5.5.0
- Operating system: Ubuntu 14.04 LTS

[12] This is Ubuntu Server GCC. The Advance Toolchain GCC is not supported by CUDA at the time of this writing.
[13] That is, Version 7 Release 1, Service Refresh 2 Fix Pack 0.

This section also covers the following prerequisites for performing the operating system installation on POWER8: Intelligent Platform Management Interface (IPMI)

Additionally, it describes the configuration of CPU Frequency Scaling

Intelligent Platform Management Interface

The following steps consist of setting an IPMI password for the system, and instructions to manage the system using IPMI.

For performing the steps described in this section, access the system's Advanced System Management Interface (ASMI) as administrator:

1. Point your web browser to the system's Flexible Service Processor (FSP) address:

   `https://`*`fsp-address`*

2. Log in as administrator (`User ID: admin`).

Steps to set the IPMI password

The following steps are used to set the IPMI password:

1. Expand **Login Profile** and click **Change Password**.
2. From the **User ID to change** list, select **IPMI**.
3. In the **Current password for user ID admin** field, enter the administrator password.
4. In the **New password for user** and **New password again** fields, enter the IPMI password.
5. Click **Continue**.

Managing the system using IPMI

The IPMItool is a command-line interface program for managing IPMI devices using the network. For example, the interface can be used to power a system on and off, and access its console[14].

You can install the IPMItool from your workstation Linux distribution packages. For example, in Ubuntu:

```
$ sudo apt-get install ipmitool
```

Or you can also build it from the source, which is available at the following site:

http://sourceforge.net/projects/ipmitool

Use the IPMItool version 1.8.14 or later, which has the latest fixes and improvements. You can check the version with the command:

```
$ ipmitool -V
```

The following list contains common **`ipmitool`** commands:

- **Power on:**

  ```
  $ ipmitool -I lanplus -H fsp-address -P ipmi-password power on
  ```

- **Power off**:

  ```
  $ ipmitool -I lanplus -H fsp-address -P ipmi-password power off
  ```

[14] Also known as Serial over LAN (SOL) session.

- **Power cycle (Power off, then Power on)**:

  ```
  $ ipmitool -I lanplus -H fsp-address -P ipmi-password power cycle
  ```
- **Power status**:

  ```
  $ ipmitool -I lanplus -H fsp-address -P ipmi-password power status
  ```
- **Open console session**:

  ```
  $ ipmitool -I lanplus -H fsp-address -P ipmi-password sol activate
  ```
- **Close console session**:

  ```
  $ ipmitool -I lanplus -H fsp-address -P ipmi-password sol deactivate
  ```

  You can also close the current console session with the following keystrokes:

  – On a non-SSH connection:

    `Enter, ~ (tilde`[15]`), . (period)`

    Note: This command might close a Secure Shell (SSH) connection, which would leave the console session open.

  – On an SSH connection:

    `Enter, ~ (tilde), ~ (tilde), . (period)`

    Note: This command leaves the SSH connection open, and closes the console session.
- **Reset the FSP**:

  $ ipmitool -I lanplus -H *fsp-address* -P *ipmi-password* bmc reset cold
- **Sensors values**:

  ```
  $ ipmitool -I lanplus -H fsp-address -P ipmi-password sdr list full
  ```
- **Specific sensor value**:

  ```
  $ ipmitool -I lanplus -H fsp-address -P ipmi-password sdr entity XY[.Z]
  ```
  [16]

Note: The IPMItool might not be available for some platforms. For an alternative IPMI utility (not covered in this publication), refer to the IPMI Management Utilities project at this link:

http://sourceforge.net/projects/ipmiutil

Ubuntu Server

The supported operating system for CUDA 5.5 on POWER8 is Ubuntu Server 14.10 in non-virtualized mode.

Note: The supported operating systems for CUDA 7.0 on POWER8 now include Ubuntu Server 14.04.2 LTS, in addition to Ubuntu Server 14.10 (both in non-virtualized mode).

To install Ubuntu Server on the IBM Power system S824L, you can choose one of the following methods:

- CD/DVD installation
- Network installation (netboot)

[15] On keyboards with dead-keys (some non-english languages), the tilde mark requires 2 keystrokes: tilde and space.

[16] Sensor entity: `XY` (entity ID) or `XY.Z` (entity ID and instance ID); listed with the `sdr elist full` IPMI command (see the fourth field of its output).

Both methods use an IPMI console session for performing the installation process.

Petitboot (system bootloader) is used to boot the installation media. Some basic instructions for operating it are as follows:

- To move the selection between elements, use the up/down or tab/shift-tab keys
- To select an entry, confirm (“click”) a button, and mark a check box, move the selection to it and press the Enter or Space key
- To return from a window (discard changes), press the Esc key
- To enter long text strings, you can use the “paste” feature of your terminal
- The bottom lines display some shortcut keys/help

Power on and open the console session

The following steps are used to power on and open the console session:

1. Power on (or power cycle, if already on) the system:

   ```
   $ ipmitool -I lanplus -H fsp-address -P ipmi-password power on # or cycle
   ```

2. Open a console session:

   ```
   $ ipmitool -I lanplus -H fsp-address -P ipmi-password sol activate
   ```

3. Wait a few minutes for the *Petitboot* window.

CD/DVD installation

The following steps consist of downloading an Ubuntu Server installation ISO image, burning it to a CD/DVD disc, and booting the CD/DVD disc:

1. Download the Ubuntu Server installation image (`ubuntu-version-server-ppc64el.iso`), which can be found at[17]:

   `http://cdimage.ubuntu.com/releases/` → *version* → `release`

   or

   `http://ubuntu.com` → `Download` → `Server` → `POWER8` → `Ubuntu Server` *version*

2. Burn (write) the ISO image to a CD/DVD disc.

   General instructions are available at:

   `http://help.ubuntu.com/community/BurningIsoHowto`

   Example 2 shows how to burn the ISO image to a CD/DVD disc in the command-line interface.

Example 2 Burning an ISO to CD/DVD disc using the wodim command-line program

```
$ dmesg | grep writer # check the CD/DVD writer device (usually /dev/sr0)
$ sudo apt-get install wodim
$ sudo wodim -v dev=/dev/sr0 ubuntu-version-server-ppc64el.iso
```

3. Insert the CD/DVD disc into the system’s optical drive.
4. In the **Petitboot** window, select **Rescan devices**.
5. Wait a few seconds for a new entry (from the Ubuntu Server CD/DVD disc).
6. Select the new entry.
7. Wait for the installer to start, and proceed with the Ubuntu Server installation.

[17] Direct link (version 14.10 “Utopic Unicorn”): `http://cdimage.ubuntu.com/releases/14.10/release/ubuntu-14.10-server-ppc64el.iso`

Network installation (netboot)

The following steps consist of configuring and testing the network, and booting the network installation files.

Network configuration

To perform the network configuration:

1. In the **Petitboot** window, select **System Configuration** (this takes you to the **Petitboot System Configuration** window).
2. From the **Network** list, select one of the following network configuration options:
   - **DHCP on all active interfaces**
   - **DHCP on a specific interface**
   - **Static IP configuration**
3. Provide the network settings in the required fields below the **Network** list, if any.

   Note: The DNS server field is required if the DNS servers are not provided by DHCP.

   - DHCP on all active interfaces: **None**
   - DHCP on a specific interface: **Device**
   - Static IP configuration: **Device, IP/mask, Gateway, DNS server(s)**
4. Select **OK**.

Network configuration test

The following steps are used to perform the network configuration test:

1. In the **Petitboot** window, select **Exit to shell** (this takes you to the Petitboot shell).
2. Perform a network test (see Example 3 for a successful *ping* test).

Example 3 Network configuration test in the Petitboot shell

```
# ping -c1 ubuntu.com
PING ubuntu.com (91.189.94.40): 56 data bytes
64 bytes from 91.189.94.40: seq=0 ttl=47 time=73.004 ms

--- ubuntu.com ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 73.004/73.004/73.004 ms
```

3. Enter the **`exit`** command to return to the Petitboot window.

```
# exit
```

If the network configuration test is successful, proceed to the next steps. Otherwise, repeat the network configuration and test steps, and verify your network settings.

Tip: You can use normal Linux network commands (for example, **`ping`**, **`ip`**, **`ifconfig`**, **`route`**) in the Petitboot shell in order to debug network configuration issues.

New entry for netboot files

The netboot files are the kernel (`vmlinux` file) and initial RAM disk (`initrd.gz` file). You will need the URL (link address) for both, which can be obtained at[18]:

`http://cdimage.ubuntu.com/netboot/` → *`version`* → `ppc64el` → `ubuntu-installer` → `ppc64el`

or:

`http://ubuntu.com` → `Download` → `Server` → `POWER8` → `Netboot image` *`version`* → `ubuntu-installer` → `ppc64el`

1. Copy the URLs for the kernel (`vmlinux` file) and initial RAM disk (`initrd.gz` file).
2. In the **Petitboot** window, press the **n** key to create a new entry (this takes you to the **Petitboot Option Editor** window).
3. From the **Device** list, select **Specify paths/URLs manually**.
4. In the **Kernel** field, enter the URL for *vmlinux*:

   `http://ports.ubuntu.com/ubuntu-ports/dists/utopic-updates/main/installer-ppc64el/current/images/netboot/ubuntu-installer/ppc64el/vmlinux`
5. In the **Initrd** field, enter the URL for *initrd.gz*:

   `http://ports.ubuntu.com/ubuntu-ports/dists/utopic-updates/main/installer-ppc64el/current/images/netboot/ubuntu-installer/ppc64el/initrd.gz`
6. Optional: If you have a `preseed`[19] file (used for automatic installation), enter the related options in the **Boot arguments** field.
7. Select **OK** (this takes you back to the **Petitboot** window).
8. Select **User item 1** (new entry).
9. The bottom line changes to:

   ```
   Info: Booting <url>
   ```
10. Wait for the files to download.
11. The bottom lines change to:

    ```
    The system is going down NOW!
    Sent SIGTERM to all processes
    Sent SIGKILL to all processes
    ```
12. Wait for the installer to start, and proceed with the Ubuntu Server installation.

Ubuntu Server installer

The Ubuntu Server installation process is standard, regardless of the GPU cards or IBM Power Systems S824L. Refer to the Ubuntu Server installation documentation for instructions:

- Overview

  `http://help.ubuntu.com` → `Ubuntu` *`version`* → `Ubuntu Server Guide` → `Installation`
- Ubuntu Installation Guide:

  `http://help.ubuntu.com` → `Ubuntu` *`version`* → `Installing Ubuntu` → `IBM/Motorola PowerPC`[20]

[18] Direct link (version 14.10 "Utopic Unicorn"): `http://ports.ubuntu.com/ubuntu-ports/dists/utopic-updates/main/installer-ppc64el/current/images/netboot/ubuntu-installer/ppc64el`

[19] For more information, check Ubuntu Installation Guide, Appendix B: Automating the installation using preseeding.

[20] This documentation is more targeted at PowerPC desktop systems, but provides some general instructions.

During installation, in the **Software selection** window, select **OpenSSH Server** to access the system through SSH upon first boot.

After the installation is finished, the system reboots. See the next section for verifying and configuring the autoboot settings.

Autoboot configuration

By default, Petitboot will automatically boot (autoboot) from any disk/network interface within 10 seconds. That is reasonable if only one operating system is installed, but you might want to change/configure it depending on your preferences:

1. In the **Petitboot** window, select **System configuration** (this takes you to the **Petitboot System Configuration** window).
2. From the **Autoboot** list, select one of the following autoboot options:
   - **Do not autoboot** (wait for user selection)
   - **Autoboot from any disk/network device** (default)
   - **Only Autoboot from a specific disk/network device**
3. Provide values for the required fields below the **Autoboot** list, if any.
   - Only Autoboot from a specific disk/network device: Select the device from the disk/net list

   > **Note:** You can check the disk devices with operating systems detected in the Petitboot window.

4. Select **OK**.

   Petitboot does not automatically boot any entry now because of user interaction. This time, it is necessary to boot the Ubuntu Server manually.
5. Select **Ubuntu**.
6. Wait for the operating system to boot (login prompt).

Verifying the network connectivity

The network settings are inherited from the installation process. Therefore, networking should be working at this point. You can verify the network connectivity with the following commands.

From another system:

- Ping the system:

  ```
  $ ping -c1 system-address
  ```
- Open an SSH connection:

  ```
  $ ssh user@system-address
  ```

If the ping and SSH connection tests are successful, proceed to the next steps. Otherwise, verify and configure the system's network settings, and restart the network interface (Example 4).

Example 4 The network configuration file, its manual page, and a network interface restart

```
$ sudo nano /etc/network/interfaces
$ man interfaces
$ sudo ifdown ethX
$ sudo ifup ethX
```

Close the console session

If you can open SSH connections to the system, it is possible to perform the next steps using SSH, which is more convenient. It is a good practice to close the IPMI console session.

In order to close the IPMI console session, use one of the methods that are described in "Intelligent Platform Management Interface" on page 15.

CUDA toolkit

In order to install the CUDA toolkit, perform the following steps:

1. Verify that the GPU cards are detected (Example 5 shows two GPU cards).

Example 5 Verifying the GPU cards with lspci

```
$ lspci | grep -i nvidia
0002:01:00.0 3D controller: NVIDIA Corporation GK110BGL [Tesla K40m] (rev a1)
0006:01:00.0 3D controller: NVIDIA Corporation GK110BGL [Tesla K40m] (rev a1)
```

2. Install basic development packages (CUDA toolkit dependencies):

   ```
   $ sudo apt-get install build-essential
   ```

3. Download the DEB package of the CUDA repository for Ubuntu 14.10 on POWER8, which is available in the Downloads section of the NVIDIA CUDA Zone website:

   `http://developer.nvidia.com/cuda-downloads-power8 → Ubuntu 14.10 DEB`

   > **Note:** On future CUDA releases, the mentioned files should be available in the normal NVIDIA CUDA downloads page, under the Linux POWER8 tab:
   >
   > `http://developer.nvidia.com/cuda-downloads → Linux POWER8.`

   ```
   $ wget
   http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1410/ppc64el/cuda
   -repo-ubuntu1410_5.5-54_ppc64el.deb
   ```

4. Install the DEB package:

   ```
   $ sudo dpkg -i cuda-repo-ubuntu1410_5.5-54_ppc64el.deb
   ```

   It inserts the CUDA repository in the package manager configuration file (Example 6).

Example 6 Package manager configuration file for the CUDA repository

```
$ cat /etc/apt/sources.list.d/cuda.list
deb http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1410/ppc64el /
```

5. Update the package manager's definitions:

   ```
   $ sudo apt-get update
   ```

6. Finally, install the CUDA toolkit using the *cuda* meta-package (Example 7 on page 22).

   The *cuda* meta-package makes the appropriate packages of the CUDA toolkit version available for the architecture, and their dependencies.

   Among other steps, the installation builds and configures the kernel modules for the NVIDIA graphics driver (which requires the basic development tools that were installed previously).

The support for Dynamic Kernel Module Support (DKMS) is installed by default (specified as a dependency of the `nvidia-340` package), so that the kernel modules are automatically rebuilt on kernel upgrades if required.

Example 7 Installation of the cuda meta-package

```
$ sudo apt-get install cuda
<...>
The following NEW packages will be installed:
  acpid cuda cuda-5-5-power8 cuda-command-line-tools-5-5-power8 <...>
  cuda-cusparse-5-5-power8 cuda-cusparse-dev-5-5-power8 <...>
  cuda-misc-headers-5-5-power8 cuda-npp-5-5-power8 <...>
  dkms libcuda1-340 nvidia-340 nvidia-340-dev nvidia-340-uvm
<...>
Do you want to continue? [Y/n] y
<...>
Loading new nvidia-340-340.50 DKMS files...
First Installation: checking all kernels...
Building only for 3.16.0-28-generic
Building for architecture ppc64el
Building initial module for 3.16.0-28-generic
Done.
<...>
DKMS: install completed.
<...>
*********************************************************************************
Please reboot your computer and verify that the nvidia graphics driver is loaded.
If the driver fails to load, please use the NVIDIA graphics driver .run installer
to get into a stable state.
*********************************************************************************
<...>
```

7. Configure the search paths for the CUDA commands and libraries, making the setting available to all users and persistent across reboots (Example 8).

Example 8 Configuration of search paths for CUDA commands and libraries

```
$ echo 'export PATH=$PATH:/usr/local/cuda-5.5-power8/bin' | sudo tee
/etc/profile.d/cuda.sh
$ echo /usr/local/cuda-5.5-power8/lib64 | sudo tee /etc/ld.so.conf.d/cuda.conf
$ sudo ldconfig
```

Note: The changes to the command search path environment variable (PATH) have no effect in the current shell (the shared library-related changes do). To apply it to the current shell run the following command:

```
$ source /etc/profile.d/cuda.sh
```

If the shared library search path is not configured correctly, CUDA applications linked to CUDA shared libraries fail, as shown in Example 9.

Example 9 Failure of CUDA applications with misconfigured shared library search path

```
<application>: error while loading shared libraries: <cuda library>.so.x.x:
cannot open shared object file: No such file or directory
```

Note: Refer to the Linux Getting Started Guide (also known as the NVIDIA CUDA Getting Started Guide for Linux) → Installing CUDA Development Tools → Install the NVIDIA CUDA Toolkit →Package Manager Installation → Environment Setup, for a different approach:

http://developer.nvidia.com/cuda-downloads

```
$ export PATH=$PATH:/usr/local/cuda-5.5-power8/bin
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda-5.5-power8/lib64
```

These are useful export entries if you do not have superuser privileges in the system. The downside is that this is not available to all users and this is not permanent across reboots. Also, the shared library search path may be inadvertently overridden when building/running non-system wide installed software.

In order to make this setting permanent across reboots, for some users, they must add these lines to their ~/.bashrc file.

8. Reboot the system:

```
$ sudo reboot
```

9. After rebooting, verify that the *nvidia* module is loaded and the GPU devices are available (Example 10).

Note: The kernel message `nvidia: module verification failed` is not an error. The message refers to loading modules with a proprietary license (see lines above it).

Example 10 Verification of `nvidia` *module and GPU status*

```
$ dmesg | grep -i nvidia
[<...>] nvidia: module license 'NVIDIA' taints kernel.
[<...>] nvidia: module license 'NVIDIA' taints kernel.
[<...>] nvidia: module verification failed: signature and/or  required key missing
- tainting kernel
[<...>] nvidia 0002:01:00.0: enabling device (0140 -> 0142)
[<...>] nvidia 0006:01:00.0: enabling device (0140 -> 0142)
[<...>] [drm] Initialized nvidia-drm 0.0.0 20130102 for 0002:01:00.0 on minor 0
[<...>] [drm] Initialized nvidia-drm 0.0.0 20130102 for 0006:01:00.0 on minor 1
[<...>] NVRM: loading NVIDIA UNIX ppc64le Kernel Module  340.50 <...>

$ lsmod | grep nvidia
nvidia              13941291  0
drm                   382761  2 nvidia

$ nvidia-smi --list-gpus
GPU 0: Tesla K40m (UUID: GPU-9c8168e0-dc96-678f-902f-3997a6f2dfc5)
GPU 1: Tesla K40m (UUID: GPU-6d42c48f-2d92-c254-9703-7218aa15fa40)
```

10. Perform a simple test with the CUDA sample *simpleCUFFT*, which uses CUDA libraries and GCC (default compiler), as shown in Example 11 on page 24[21].

[21] Shell linebreaks added for clarity.

Example 11 Building and running simpleCUFFT with GCC

```
$ cp -r /usr/local/cuda-5.5-power8/samples/ ~/cuda-samples
$ cd ~/cuda-samples/7_CUDALibraries/simpleCUFFT/
$ make
/usr/local/cuda-5.5-power8/bin/nvcc -ccbin g++ -I../../common/inc  -m64
-gencode arch=compute_20,code=sm_20 -gencode arch=compute_30,code=sm_30 -gencode
arch=compute_35,code=\"sm_35,compute_35\" -o simpleCUFFT.o -c simpleCUFFT.cu
/usr/local/cuda-5.5-power8/bin/nvcc -ccbin g++   -m64        -o simpleCUFFT
simpleCUFFT.o  -lcufft
mkdir -p ../../bin/ppc64le/linux/release
cp simpleCUFFT ../../bin/ppc64le/linux/release

$ ../../bin/ppc64le/linux/release/simpleCUFFT
[simpleCUFFT] is starting...
GPU Device 0: "Tesla K40m" with compute capability 3.5

Transforming signal cufftExecC2C
Launching ComplexPointwiseMulAndScale<<< >>>
Transforming signal back cufftExecC2C
```

For more information, refer to the Linux Getting Started Guide (also known as NVIDIA CUDA Getting Started Guide for Linux), available in the Downloads section of the NVIDIA CUDA Zone website:

http://developer.nvidia.com/cuda-downloads

IBM XL C/C++ Compiler

To install the IBM XL C/C++ Compiler, perform the following steps:

1. Install the basic development packages (IBM XL C/C++ Compiler dependencies):

   ```
   $ sudo apt-get install build-essential
   ```

2. Install the DEB packages provided with your installation media (Example 12).

Example 12 Installation of the XL C/C++ Compiler packages

```
$ cd <path to IBM XL C/C++ deb-packages>

$ sudo dpkg -i xlc*.deb libxlc*.deb libxlmass*.deb libxlsmp*.deb # or *.deb
<...>
Setting up xlc-license.13.1.1 (13.1.1.0-141105) ...
Setting up libxlc (13.1.1.0-141105) ...
Setting up libxlc-devel.13.1.1 (13.1.1.0-141105) ...
Setting up libxlmass-devel.8.1.0 (8.1.0.0-141027) ...
Setting up libxlsmp (4.1.0.0-141010) ...
Setting up libxlsmp-devel.4.1.0 (4.1.0.0-141010) ...
Setting up xlc.13.1.1 (13.1.1.0-141105) ...
Please run 'sudo /opt/ibm/xlC/13.1.1/bin/xlc_configure' to review the license and
configure the compiler.
```

3. To review the license and configure the compiler, enter:

   ```
   $ sudo /opt/ibm/xlC/13.1.1/bin/xlc_configure
   ```

   Note: For non-interactive installations, see Example 13 on page 25.

Example 13 Reviewing the license and configuring the IBM XL C/C++ Compiler non-interactively

```
$ echo 1 | sudo /opt/ibm/xlC/13.1.1/bin/xlc_configure
<...>
International License Agreement <...>
<...>
Press Enter to continue viewing the license agreement, or, Enter "1" to accept the
agreement, "2" to decline it or "99" to go back to the previous
screen, "3" Print.
[INPUT] ==> 1
<...>
update-alternatives: using /opt/ibm/xlC/13.1.1/bin/xlc to provide /usr/bin/xlc
(xlc) in auto mode
<...>
INFORMATIONAL: GCC version used in
"/opt/ibm/xlC/13.1.1/etc/xlc.cfg.ubuntu.14.04.gcc.4.8.2" -- "4.8.2"
INFORMATIONAL: /opt/ibm/xlC/13.1.1/bin/xlc_configure completed successfully
```

4. Perform a simple test with the CUDA sample *simpleCUFFT*, which uses CUDA libraries and IBM XL C++ (using the **`-ccbin xlC`** option for **`nvcc`**, which can be defined in CUDA samples with the make variable `GCC`), as shown in Example 14.

   A few points:

   – The compiler warning is not an error; it refers to analysis of the source code.
   – The output of running *simpleCUFFT* is the same as shown in Example 11 on page 24[22].

Example 14 Building and running simpleCUFFT with IBM XL C++

```
Note: this step may have been performed in the other example (GCC).
$ cp -r /usr/local/cuda-5.5-power8/samples/ ~/cuda-samples
$ cd ~/cuda-samples/7_CUDALibraries/simpleCUFFT/

$ make clean
rm -f simpleCUFFT.o simpleCUFFT
rm -rf ../../bin/ppc64le/linux/release/simpleCUFFT

$ make GCC=xlC
/usr/local/cuda-5.5-power8/bin/nvcc -ccbin xlC -I../../common/inc  -m64
-gencode arch=compute_20,code=sm_20 -gencode arch=compute_30,code=sm_30 -gencode
arch=compute_35,code=\"sm_35,compute_35\" -o simpleCUFFT.o -c simpleCUFFT.cu
../../common/inc/helper_cuda.h:493:9: warning: 4 enumeration values not handled in
switch: 'CUFFT_INCOMPLETE_PARAMETER_LIST', 'CUFFT_INVALID_DEVICE',
      'CUFFT_PARSE_ERROR'... [-Wswitch]
switch (error)
        ^
1 warning generated.
/usr/local/cuda-5.5-power8/bin/nvcc -ccbin xlC   -m64        -o simpleCUFFT
simpleCUFFT.o  -lcufft
mkdir -p ../../bin/ppc64le/linux/release
cp simpleCUFFT ../../bin/ppc64le/linux/release

$ ../../bin/ppc64le/linux/release/simpleCUFFT
<...>
```

[22] Shell linebreaks added for clarity.

For more information, refer to the IBM Knowledge Center website:

`http://ibm.com/support/knowledgecenter → Rational → C and C++ Compilers → XL C/C++ for Linux → XL C/C++ for Linux` *`version`* `→ Installation Guide`

Java

To install the IBM Developer Kit for Linux, Java Technology Edition, perform the following steps:

1. Download the IBM Developer Kit for Linux, Java Technology Edition, which is available at:

   `http://ibm.com/developerworks/java/jdk/ → in row Linux, Downloads → Java SE Version 7 → in row 64-bit IBM POWER (LE), Download Now`

   Note: Download the SDK file *ibm-java-ppc64le-sdk-<version>.bin*. (This is not the SDK in the `archive` format, and not the JRE only, because JRE is included in the SDK.)

2. Make the file executable:

   ```
   $ chmod +x ibm-java-ppc64le-sdk-7.1-2.0.bin
   ```

3. Perform the installation with one of the following methods:

   – Interactive: To review the license and options:

     ```
     $ sudo ./ibm-java-ppc64le-sdk-7.1-2.0.bin
     ```

   – Non-interactive: Using default options:

     ```
     $ sudo ./ibm-java-ppc64le-sdk-7.1-2.0.bin -i silent
     ```

4. Configure the environment and command search path for Java (Example 15)[23].

Example 15 Configuring the environment and command search path for Java

```
$ cat <<"EOF" | sudo tee /etc/profile.d/ibm-java.sh
export JAVA_HOME=/opt/ibm/java-ppc64le-71
export PATH=$PATH:$JAVA_HOME/bin:$JAVA_HOME/jre/bin
EOF
```

Note: The changes to the shell environment (PATH) have no effect in the current shell. To apply the changes to the current shell, run:

```
$ source /etc/profile.d/ibm-java.sh
```

5. Verify the Java virtual machine and Java Compiler commands (Example 16).

Example 16 Verifying the Java virtual machine and Java Compiler commands

```
$ java -version
java version "1.7.0"
Java(TM) SE Runtime Environment (build pxl6470_27sr2-20141101_01(SR2))
IBM J9 VM (build 2.7, JRE 1.7.0 Linux ppc64le-64 Compressed References
20141031_220034 (JIT enabled, AOT enabled)
J9VM - R27_Java727_SR2_20141031_1950_B220034
JIT  - tr.r13.java_20141003_74587.02
GC   - R27_Java727_SR2_20141031_1950_B220034_CMPRSS
J9CL - 20141031_220034)
```

[23] The first EOF delimiter is quoted (“EOF”) to avoid variable expansion.

```
JCL - 20141004_01 based on Oracle 7u71-b13

$ javac -version
javac 1.7.0-internal
```

6. Perform a simple test that uses *com.ibm.gpu* classes (discussed in "CUDA and IBM Java" on page 40) to list the available GPUs (Example 17 shows two GPUs).

   Notes about the **java** command options:

   – *-Xmso512k*: Java requires an operating system thread stack size of at least 300 KB for CUDA functionality (otherwise, an exception is triggered), as of this writing
   – *-Dcom.ibm.gpu.enable*: Enables CUDA functionality
   – *-Dcom.ibm.gpu.verbose*: Verbose logging of the CUDA functionality

Example 17 Java application with the com.ibm.gpu classes to list the available GPUs

```
$ cat > ListGPUs.java <<EOF
import com.ibm.gpu.*;

public class ListGPUs {
   public static void main(String[] args) throws Exception {
      System.out.println(CUDAManager.getInstance().getCUDADevices());
   }
}
EOF

$ javac ListGPUs.java

$ java -Xmso512k -Dcom.ibm.gpu.enable ListGPUs
[com.ibm.gpu.CUDADevice@5530ad1, com.ibm.gpu.CUDADevice@611a71b1]

$ java -Xmso512k -Dcom.ibm.gpu.enable -Dcom.ibm.gpu.verbose ListGPUs
[IBM GPU]: [<...>]: System property com.ibm.gpu.enable=
[IBM GPU]: [<...>]: Enabling sort on the GPU
[IBM GPU]: [<...>]: Discovered 2 devices
[IBM GPU]: [<...>]: Providing identifiers of discovered CUDA devices, found: 2
devices
[IBM GPU]: [<...>]: Discovered devices have the following identifier(s):
0, 1
[com.ibm.gpu.CUDADevice@5530ad1, com.ibm.gpu.CUDADevice@611a71b1
```

For more information about the development of Java applications that use GPUs, refer to the IBM Knowledge Center website:

`http://ibm.com/support/knowledgecenter` → `WebSphere` → `IBM SDK, Java Technology Edition` → `IBM SDK, Java Technology Edition` *`version`* → `Developing Java applications` → `Writing Java applications that use a graphics processing unit`

For more information about the installation and configuration of the IBM JDK, refer to the IBM Knowledge Center website:

`http://ibm.com/support/knowledgecenter` → WebSphere → IBM SDK, Java Technology Edition → IBM SDK, Java Technology Edition *version* → Linux User Guide for IBM SDK, Java Technology Edition, *version* → Installing and configuring the SDK and Runtime Environment

Also see the IBM developerWorks® pages for IBM Developer Kit for Linux, Java Technology Edition:

`http://ibm.com/developerworks/java/jdk/linux`

Note: The terms "IBM Developer Kit for Linux, Java Technology Edition" and "IBM SDK, Java Technology Edition" are used interchangeably in this section.

CPU frequency scaling

Linux provides CPU frequency scaling with the `cpufreq` mechanism. It can adjust the processors' clock frequency over time, according to certain policies, called *governors*.

The default scaling governor in Ubuntu Server is `ondemand`, which dynamically adjusts CPU frequency according to system load, on a processor-core level. This enables power-savings without compromising performance. A core's frequency is maintained low when idle, and increased under load.

On highly-utilized HPC systems, the preferable scaling governor is `performance`, which statically sets CPU frequency to the highest available. This ensures that a core is clocked at maximal frequency at all times, and eliminates any frequency-scaling overhead and jitter.

The cpufreq settings are per-CPU (hardware thread) files placed in:

`/sys/devices/system/cpu/cpu<`*`number`*`>/cpufreq/`

Although the `cpufreq` settings are exposed *per-CPU* (or *per-hardware thread)*, they are actually *per-core* on POWER8 (that is, equal among all hardware threads in a core). This is reflected in the `related_cpus` file, as seen in Example 18, which shows that CPU (or hardware thread) 0 shares settings with other CPUs (hardware threads) of its core (its eight hardware threads, in SMT8 mode).

Example 18 The `related_cpus` file

```
$ cat /sys/devices/system/cpu/cpu0/cpufreq/related_cpus
0 1 2 3 4 5 6 7
```

As an example, the current scaling governor can be displayed/changed by using the `scaling_governor` file, and the available scaling governors are displayed in the `scaling_available_governors` file (Example 19).

Example 19 Accessing the `scaling_governor` and `scaling_available_governors` files

```
$ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
ondemand

$ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_governors
conservative userspace powersave ondemand performance
```

```
$ echo performance | sudo tee
/sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
performance

$ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
performance
```

Set the scaling governor to performance on all CPUs

The `cpufrequtils` package provides tools for managing `cpufreq` settings on all CPUs/hardware-threads (system-wide), and making changes persistent across reboots.

1. Install `cpufrequtils`:

   ```
   $ sudo apt-get install cpufrequtils
   ```

2. Add the line `GOVERNOR=”performance”` to the file `/etc/default/cpufrequtils`. This makes the setting persistent across reboots. You can either use a text editor or the following command:

   ```
   $ echo ‘GOVERNOR=”performance”’ | sudo tee -a /etc/default/cpufrequtils
   ```

3. Disable the system’s default ondemand governor setting:

   ```
   $ sudo update-rc.d ondemand disable
   ```

4. Finally, activate the `performance` governor without rebooting the system:

   ```
   $ sudo service cpufrequtils start
   ```

You can use the **`cpufreq-info`** command for obtaining status of the `cpufreq` settings. See Example 20 to verify the number of CPUs/hardware-threads that are running with a certain governor.

Example 20 Number of CPUs with respective scaling governor

```
$ cpufreq-info | grep ‘The governor’ | uniq -c
   160           The governor “performance” <...>
```

CUDA cross-platform development

The CUDA cross-platform development tools run in a x86-64 computer, and the CUDA applications run on the POWER8 system (remotely). The supported operating system is Ubuntu 14.04 LTS for amd64.

Note: Installation instructions for Ubuntu 14.04 LTS for amd64 are not covered in this paper. Refer to the Ubuntu Installation Guide at:

`http://help.ubuntu.com` → **`Ubuntu 14.04 LTS`** → **`Installing Ubuntu`** → **`AMD64 & Intel EM64T.`**

You may download Ubuntu 14.04 LTS for amd64, at:

`http://ubuntu.com` → **`Download`** → **`Desktop`** → **`Ubuntu 14.04(.x) LTS`** → **`Flavour: 64-bit`** → **`Download.`**

To install the cross-platform development packages, it is required to add the target system’s architecture as a foreign architecture in the package manager. This is a requirement for the CUDA packages.

For CUDA cross-platform development in POWER8 systems, add the `ppc64el` architecture (the architecture string for the 64-bit PowerPC instruction-set in little-endian mode in Ubuntu and Debian distributions) as a foreign architecture for the package manager (`dpkg`). This does not interfere with the primary architecture of the package manager, which is still `amd64` (similarly, for the 64-bit version of the x86 instruction-set). The user might observe that `i386` (similarly, for the x86 instruction-set) is already a foreign architecture by default, which enables the installation of 32-bit packages. Refer to Example 21.

Example 21 Adding ppc64el as a foreign architecture in Ubuntu 14.04 LTS for amd64

```
Primary architecture:
$ dpkg --print-architecture
amd64

Foreign architectures:
$ dpkg --print-foreign-architectures
i386

$ sudo dpkg --add-architecture ppc64el

$ dpkg --print-foreign-architectures
i386
ppc64el
```

Proceed to install the CUDA repository package (which inserts the CUDA repository in the package manager configuration files), and update the package manager's definitions (Example 22).

Example 22 Installing the CUDA repository package

```
$ wget
http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1404/x86_64/cuda-rep
o-ubuntu1404_5.5-54_amd64.deb

$ sudo dpkg -i cuda-repo-ubuntu1404_5.5-54_amd64.deb

$ cat /etc/apt/sources.list.d/cuda.list
deb http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1404/x86_64 /

$ sudo apt-get update
<...>
```

Note: During the **`apt-get update`** command, there are several warnings for not found files, which can be safely ignored. For example:

```
W: Failed to fetch <...>/binary-ppc64el/Packages  404  Not Found [IP: <...>]
```

This warning happens as a result of the foreign-architecture setting combined with a different URL used for Ubuntu servers hosting ppc64el binary packages (that is, `ports.ubuntu.com`), at the time this publication was written.

Finally, install the `cuda-cross-ppc64el` meta-package, and the `nvidia-nsight` packages (Example 23 on page 31).

Example 23 Installing the cuda-cross-ppc64el meta-package and the nvidia-nsight package

```
$ sudo apt-get install cuda-cross-ppc64el
<...>
The following NEW packages will be installed:
<...>
  cuda-core-5-5-power8 cuda-cross-ppc64el cuda-cross-ppc64el-5-5-power8
<...>
Do you want to continue? [Y/n] y
WARNING: The following packages cannot be authenticated!
<...>
Install these packages without verification? [y/N] y
<...>
Setting up cuda-cross-ppc64el (5.5-54) ...
*** LICENSE AGREEMENT ***
<...>

$ sudo apt-get install nvidia-nsight
<...>
The following NEW packages will be installed:
  nvidia-cuda-doc nvidia-nsight
<...>
Do you want to conitnue? [Y/n] y
<...>
```

For more information, refer to the Linux Getting Started Guide (also known as NVIDIA CUDA Getting Started Guide for Linux) → Installing CUDA Development Tools → Install the NVIDIA CUDA Toolkit Package → Manager Installation → Available packages:

http://developer.nvidia.com/cuda-downloads

System monitoring

The nvidia-smi tool, which is provided by NVIDIA, is used to manage and monitor activities of the GPU devices. In Ubuntu 14.10, the nvidia-smi tool is included with the nvidia-340 DEB package. Because nvidia-smi has many options, the topic is only introduced in this section. It is recommended that you learn more about the nvidia-smi tool at the following site:

http://developer.nvidia.com/nvidia-system-management-interface

Querying the state and properties of devices

Run **nvidia-smi** to show a summary of all GPUs attached to the system. Use **nvidia-smi -q** to display the current state and properties of the devices (Example 24). An individual device can be accessed with the **-i <id>** option, where *<id>* is the identification number of the card.

Example 24 nvidia-smi: displaying devices state with -q option (some output omitted)

```
$ sudo nvidia-smi -q

==============NVSMI LOG==============

Timestamp                           : Thu Nov 27 10:32:44 2014
Driver Version                      : 340.50

Attached GPUs                       : 2
```

```
GPU 0002:01:00.0
    Product Name                    : Tesla K40m
    Product Brand                   : Tesla
    Display Mode                    : Disabled
    Display Active                  : Disabled
    Persistence Mode                : Disabled
    Accounting Mode                 : Disabled
    Accounting Mode Buffer Size     : 128
    Driver Model
        Current                     : N/A
        Pending                     : N/A
    Serial Number                   : 0324114015717
    GPU UUID                        : GPU-9c8168e0-dc96-678f-902f-3997a6f2dfc5
    Minor Number                    : 0
    VBIOS Version                   : 80.80.3E.00.01
    MultiGPU Board                  : No
    Board ID                        : 0x20100
    Inforom Version
        Image Version               : 2081.0202.01.04
        OEM Object                  : 1.1
        ECC Object                  : 3.0
        Power Management Object     : N/A
    GPU Operation Mode
        Current                     : N/A
        Pending                     : N/A
    PCI
        Bus                         : 0x01
        Device                      : 0x00
        Domain                      : 0x0002
        Device Id                   : 0x102310DE
        Bus Id                      : 0002:01:00.0
        Sub System Id               : 0x097E10DE
        GPU Link Info
            PCIe Generation
                Max                 : 3
                Current             : 3
            Link Width
                Max                 : 16x
                Current             : 16x
        Bridge Chip
            Type                    : N/A
            Firmware                : N/A
    Fan Speed                       : N/A
    Performance State               : P0
    Clocks Throttle Reasons
        Idle                        : Not Active
        Applications Clocks Setting : Active
        SW Power Cap                : Not Active
        HW Slowdown                 : Not Active
        Unknown                     : Not Active
    FB Memory Usage
        Total                       : 11519 MiB
        Used                        : 55 MiB
        Free                        : 11464 MiB
    BAR1 Memory Usage
```

```
        Total                       : 16384 MiB
        Used                        : 2 MiB
        Free                        : 16382 MiB
    Compute Mode                    : Default
    Utilization
        Gpu                         : 0 %
        Memory                      : 0 %
        Encoder                     : 0 %
        Decoder                     : 0 %
    Ecc Mode
        Current                     : Enabled
        Pending                     : Enabled
    ECC Errors
        Volatile
            Single Bit
                Device Memory       : 0
                Register File       : 0
                L1 Cache            : 0
                L2 Cache            : 0
                Texture Memory      : 0
                Total               : 0
            Double Bit
                Device Memory       : 0
                Register File       : 0
                L1 Cache            : 0
                L2 Cache            : 0
                Texture Memory      : 0
                Total               : 0
        Aggregate
            Single Bit
                Device Memory       : 0
                Register File       : 0
                L1 Cache            : 0
                L2 Cache            : 0
                Texture Memory      : 0
                Total               : 0
            Double Bit
                Device Memory       : 0
                Register File       : 0
                L1 Cache            : 0
                L2 Cache            : 0
                Texture Memory      : 0
                Total               : 0
    Retired Pages
        Single Bit ECC              : 0
        Double Bit ECC              : 0
        Pending                     : No
    Temperature
        GPU Current Temp            : 39 C
        GPU Shutdown Temp           : 95 C
        GPU Slowdown Temp           : 90 C
    Power Readings
        Power Management            : Supported
        Power Draw                  : 63.16 W
        Power Limit                 : 235.00 W
        Default Power Limit         : 235.00 W
        Enforced Power Limit        : 235.00 W
        Min Power Limit             : 180.00 W
        Max Power Limit             : 235.00 W
    Clocks
        Graphics                    : 745 MHz
        SM                          : 745 MHz
        Memory                      : 3004 MHz
    Applications Clocks
        Graphics                    : 745 MHz
        Memory                      : 3004 MHz
    Default Applications Clocks
        Graphics                    : 745 MHz
        Memory                      : 3004 MHz
    Max Clocks
        Graphics                    : 875 MHz
        SM                          : 875 MHz
        Memory                      : 3004 MHz
    Clock Policy
        Auto Boost                  : N/A
        Auto Boost Default          : N/A
    Compute Processes               : None

GPU 0006:01:00.0
    Product Name                    : Tesla K40m
    Product Brand                   : Tesla
<... output omitted ...>
```

Query just one or more of the device properties by adding the **`-d <property_name>`** option, where *property_name* can assume the values that are shown in Table 2.

Table 2 nvidia-smi: query properties name

Property	Description
MEMORY	Show memory usage statistics
UTILIZATION	Show utilization statistics of GPU, memory, encoder, and decoder subsystems
ECC	Show status of ECC and error statistics
TEMPERATURE	Display GPU temperature
POWER	Show power management status
CLOCK	Show clock rate of graphics, SM, memory, and applications; display usage statistics
SUPPORTED_CLOCKS	Display available combinations to GPU and memory clock boost
COMPUTE	Display current compute mode
PIDS	List running applications
PERFORMANCE	Show performance statistics
PAGE_RETIREMENT	Display number of retired pages
ACCOUNTING	Display current accounting mode

Managing devices

Several device settings can be enabled and disabled by, respectively, passing 1 (one) or 0 (zero) to the management options of the **`nvidia-smi`** command. For example, use option **`-e`** to set/unset error correcting code (ECC) support and use **`-p`** to reset ECC errors.

To save the current properties between driver unloads, use the **`-e 1`** option to turn on persistence mode. This loads the driver even though no compute application is executing.

The GPU compute mode is set with the **`-c`** option. Allowed values are 0/DEFAULT, 1/EXCLUSIVE_THREAD, 2/PROHIBITED, and 3/EXCLUSIVE_PROCESS.

To boost performance, other nvidia-smi options allow you to manage memory and graphics clock rates for computer applications. Use options **`-ac`**, **`-rac`**, and **`-acp`** to respectively set clocks values, reset to base clocks, and permit non-root users clock changes. It is also possible to control the auto boost mechanism using **`--auto-boost-default`**, **`--auto-boost-default-force`**, and **`--auto-boost-permission`** options.

Application development

The IBM POWER8 chip is a highly parallel microprocessor with strong memory subsystem and high floating-point performance.

Moreover, POWER8 is the first IBM POWER architecture processor that supports little-endian memory access mode in virtualized and non-virtualized (or bare-metal) mode. At the time this publication was written, the IBM Power S824L included the Ubuntu Linux operating system distribution that supports little-endian.

In order to fully leverage the computational power of the IBM Power Systems S824L, you need to harness the computing capabilities of the GPU that is included with the system. The GPU acceleration can be used to offload highly parallel operations and boost Java performance.

The application developer has access to the GPU accelerator through the NVIDIA CUDA parallel computing platform and programming model. In theory, it is possible to develop applications that exploit the benefits of GPU even without the knowledge of the CUDA programming model. The CUDA toolkit includes a set of libraries that are tuned for NVIDIA GPUs.

In the area of technical computing, the NVIDIA CUDA infrastructure is well known to application developers. You can find information at the NVIDIA website:

`http://developer.nvidia.com/cuda-zone`

In the area of enterprise application development, IBM has made significant steps toward the integration of CUDA into a Java environment. Currently, IBM JDK includes Java functions that directly take advantage of the GPU. The CUDA4J application programming interface makes it possible to invoke external CUDA kernels directly from the Java code. These topics are covered in section “CUDA and IBM Java” on page 40.

The CUDA toolkit includes the Nsight Eclipse Edition integrated development environment for CUDA applications. Section “Nsight” on page 44 explains how to create, edit, build, and launch projects from within the environment.

This publication emphasizes only those CUDA programming features that are specific for POWER8 based server systems. You can find the manuals relevant to your system in the

<cuda_installation_dir>/doc/ directory of your server or cross-platform development computer.

CUDA paradigm

CUDA is a parallel computing platform and programming model by NVIDIA. When thinking about employing CUDA, you need to keep in mind the following two points:

- CUDA is about throughput, not about latency
- CUDA is for highly data-parallel applications

A POWER8 based server equipped with GPU accelerators constitutes in some sense a perfectly balanced system. The primary beneficiaries of this union are hybrid applications: the applications that have both highly data-parallel parts and latency-critical pieces that require large memory bandwidth.

The most favorable workload for GPU is the one where there is the need to execute exactly the same sequence of operations on each element of a large data set. Traditionally, these are image and video processing applications, and graphics rendering. The CUDA programming model also proved to be useful in such areas as computational biology, chemistry, physics, molecular dynamics, bioinformatics, material science, computational fluid dynamics, structural mechanics, computer-aided design, electronic design automation, seismic processing and interpretation, reservoir simulation, financial analytics, and data mining.

The CUDA application typically adheres to the following execution model:

1. Data is copied from main memory into GPU memory
2. CPU instructs GPU to schedule a processing
3. GPU computing cores execute a specified GPU kernel function
4. Data is copied back to the main memory from GPU memory

In order to better utilize the computing resources of CPU and GPU, it is better to overlap memory copy and computation operations. That is done by using CUDA asynchronous memory transfers and CUDA execution streams.

In CUDA for C/C++, the source files that need to be compiled with the CUDA compiler should have the *.cu* file extension. The source files that do not contain CUDA statements have the usual *.c* or *.cpp* file extension.

Compilers

The CUDA application includes parts that are executed solely on a CPU (host code) and pieces that are run solely on a GPU (device code). The NVIDIA CUDA Compiler Driver (NVCC) takes the source code and forwards the host code functions to the program that is specified with the **`-ccbin`** option as a host code compiler.

Based on the CUDA programming model, both GCC and the IBM XL C/C++ compiler can be used for host code compilation. The following sections demonstrate how to compile CUDA applications by using GCC and the IBM XL C/C++ compilers.

NVIDIA CUDA Compiler Driver

The NVCC is a compiler intended for use with CUDA. It is based on the widely used LLVM open source compiler infrastructure. CUDA codes run on both the CPU and GPU. NVCC separates these two parts and sends the host source code (the part of the code that runs on the CPU) to a C/C++ compiler, and sends the device source code (the part that runs on the GPU) to NVIDIA compilers/assemblers.

Any source file containing CUDA language extensions (*.cu*) must be compiled with NVCC. NVCC is a compiler driver that simplifies the process of compiling C/C++ code. It provides simple and familiar command-line options and executes them by invoking the collection of tools that implement the different compilation stages. For more information about NVCC, refer to the NVIDIA website:

`http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc`

IBM XL C/C++ Compiler as the host compiler

The IBM XL C/C++ Compiler v13.1.1 is the first version to support CUDA for POWER8 on little-endian Linux.

In this section, the CUDA sample project *simplePrintf* is used as an example (see "NVIDIA CUDA Toolkit code samples" on page 55). The project is at this location:

`<cuda_installation_dir>/samples/0_Simple/simplePrintf`

To compile the example with the IBM XL C/C++ compiler as the host compiler, copy the source code files to a directory where you have file read, write, and execute permission, and run the `make GCC=xlC` command; the output program is copied to the `release` directory (Example 25).

Example 25   Building the simplePrintf sample

```
$ cp -r /usr/local/cuda-5.5-power8/samples/ ~/cuda-samples
$ cd ~/cuda-samples/0_Simple/simplePrintf/
$ make GCC=xlC
<...>
cp simplePrintf ../../bin/ppc64le/linux/release
```

The following shows how to compile the example by manually invoking an **`nvcc`** compilation command:

```
$ nvcc -ccbin xlC -I<cuda_installation_dir>/samples/common/inc -m64 \
    -gencode arch=compute_35,code=sm_35 simplePrintf.cu -o simplePrintf
```

Example 26 shows the output of executing program *simplePrintf*.

Example 26   The output of the CUDA application example (simplePrintf)

```
GPU Device 0: "Tesla K40m" with compute capability 3.5

Device 0: "Tesla K40m" with Compute 3.5 capability
printf() is called. Output:

[0, 0]:         Value is:10
[0, 1]:         Value is:10
[0, 2]:         Value is:10
<... output omitted ...>
[2, 7]:         Value is:10
```

The IBM Power Systems S824L server supports NVIDIA Tesla K40 GPUs based on the NVIDIA Kepler architecture GK110B. For application binary compatibility, set the nvcc option **`code=sm_35`**. To enable full support for the Kepler architecture, the option **`arch=compute_35`** is also required. For more information, see the following site:

`http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#application-compatibility`

GCC compiler as the host compiler

To compile the CUDA *clock* sample with **gcc** as host compiler, go to the source code directory and run:

```
$ make
```

The sample code can also be compiled with **nvcc** on the command line as follows:

```
$ nvcc -ccbin g++ -I<cuda_installation_dir>/samples/common/inc -m64 \
    -gencode arch=compute_35,code=sm_35 -o clock.o -c clock.cu
```

The output of executing *clock* is shown in Example 27.

Example 27   The output of the CUDA application example (clock)

```
CUDA Clock sample
GPU Device 0: "Tesla K40m" with compute capability 3.5

Total clocks = 654801
```

Running CUDA applications

You can run a CUDA application from the command line as a normal application. No special action is required. But in some cases, you need to provide a path to the shared libraries (see the configuration steps discussed in “CUDA toolkit” on page 21) or control the execution by setting environment variables.

Environment variables

The execution of CUDA applications can be controlled by setting a number of environment variables. A particularly important one to highlight is the *CUDA_VISIBLE_DEVICES* environment variable. This variable helps to control which devices your application uses. CUDA applications only see those devices whose index is given in the sequence that is assigned to the variable. The devices are also enumerated in the order of the sequence.

The *deviceQuery* example from the CUDA toolkit code samples (see “NVIDIA CUDA Toolkit code samples” on page 55) helps to illustrate the effect that the *CUDA_VISIBLE_DEVICES* environment variable has on the devices available for an application. For a system with two GPU cards, the result of the *deviceQuery* execution is listed in Example 28 for various values that are assigned to the *CUDA_VISIBLE_DEVICES* environment variable.

Example 28   The effect of the `CUDA_VISIBLE_DEVICES` environment variable on device enumeration

```
$ unset CUDA_VISIBLE_DEVICES
$ ./deviceQuery | grep PCI
  Device PCI Domain ID / Bus ID / location ID:   2 / 1 / 0
  Device PCI Domain ID / Bus ID / location ID:   6 / 1 / 0
$ CUDA_VISIBLE_DEVICES=0,1 ./deviceQuery | grep PCI
  Device PCI Domain ID / Bus ID / location ID:   2 / 1 / 0
  Device PCI Domain ID / Bus ID / location ID:   6 / 1 / 0
$ CUDA_VISIBLE_DEVICES=1,0 ./deviceQuery | grep PCI
  Device PCI Domain ID / Bus ID / location ID:   6 / 1 / 0
  Device PCI Domain ID / Bus ID / location ID:   2 / 1 / 0
$ CUDA_VISIBLE_DEVICES=0 ./deviceQuery | grep PCI
  Device PCI Domain ID / Bus ID / location ID:   2 / 1 / 0
$ CUDA_VISIBLE_DEVICES=1 ./deviceQuery | grep PCI
  Device PCI Domain ID / Bus ID / location ID:   6 / 1 / 0
```

GPU accelerated libraries

The CUDA toolkit ships with a set of highly optimized GPU accelerated libraries. It is recommended that you use routines from these libraries instead of writing your own implementation of standard functions.

As part of the CUDA toolkit, you might want to use the following APIs and libraries:

CUDA math API	This module contains a set of regular mathematical functions to be used in device code (trigonometric and hyperbolic functions, error functions, logarithmic and exponential functions, rounding functions, and so on).
CUBLAS	The CUBLAS library implements Basic Linear Algebra Subprograms (BLAS) on top of CUDA. You should use BLAS level 3 functions if you want to fully leverage the computational power of your GPU.
CUFFT	The CUFFT library contains a set of routines for computing discrete fast fourier transform (FFT). It consists of two separate libraries: CUFFT and CUFFTW. The CUFFT library leverages GPU to compute FFT for complex-valued and real-valued data sets. The CUFFTW library provides the FFTW3 API to facilitate porting of existing applications that use the FFTW[24] library.
CURAND	The CURAND library provides random number generation facilities. It consists of two parts: a library on the host (CPU) side and a library on the device (GPU) side. The host-side library is used to generate a set of random numbers in host or device memory for later use. The device-side library defines random number generation functions to be used by user-written kernels. The latter approach implies that random data is immediately consumed by user kernels, and that global memory copying operations are not required.
CUSPARSE	This module includes subroutines that implement basic linear algebra operations with sparse vectors and matrixes. The library supports several storage formats for sparse matrixes.
NPP	The focus of NVIDIA Performance Primitives library is digital signal processing. The library contains a set of routines to handle imaging and video data.
Thrust	This is a C++ template library for CUDA, which was inspired by the C++ Standard Template Library (STL). The library provides a collection of common data parallel primitives (transformations, reductions, prefix-sums, reordering, and sorting). The application developer creates complex algorithms based on these primitives, and thrust automatically tries to choose the most efficient implementation.

You can access user guides for these libraries in the “CUDA API References” section of CUDA toolkit documentation (see “Application development” on page 35).

Compilation and linking

When writing a code that uses CUDA toolkit libraries, include the corresponding header files and link with the respective libraries. For example, if your program *cufft_sample.cu* depends on the CUFFT library, compile it with the following command:

```
$ nvcc -ccbin xlC -m64 cufft_sample.cu -o cufft_sample -lcufft
```

[24] See *FFTW home page,* found at: http://fftw.org

For the names of the header files and the libraries, check the “CUDA API References” section of CUDA Toolkit Documentation (see “Application development” on page 35).

Sometimes, your environment might not contain full paths to the compilers and options for the header files and libraries. In this case, explicitly specify them on the command line:

- `<`*`cuda_installation_dir`*`>/bin/nvcc`
- `<`*`ibmxl_installation_dir`*`>/bin/xlC` or `<`*`gcc_installation_dir`*`>/bin/g++`
- `-I<`*`cuda_installation_dir`*`>/include`
- `-L<`*`cuda_installation_dir`*`>/lib64`

CUDA and IBM Java

Many features of the IBM POWER8 processor were designed to deal with big data and analytics workloads. Many data processing applications are currently written in the Java programming language. If you need to accelerate such applications, offload the most computationally intensive parts to hardware accelerators. However, it is unfeasible to completely refactor such applications into CUDA C/C++.

In order to provide better experience for Java users and leverage the capabilities of GPU-accelerated POWER8 servers, IBM SDK, Java Technology Edition started to include support for offloading computations to the GPU directly from the Java code. IBM SDK, Java Technology Edition support for GPU started from Version 7 Release 1, Service Refresh 2 Fix pack 0 (7.1-2.0).

The application developer has several options for using GPU on POWER8 from the Java code:

- Let the Java virtual machine (JVM) decide when to offload processing to a GPU
- Use the *com.ibm.gpu* classes to offload specific tasks
- Use the CUDA4J API to specify in the application exactly when to use the GPU

As usual, your Java program can target a specific GPU if you set the *CUDA_VISIBLE_DEVICES* environment variable, as described in “Environment variables” on page 38.

If you experience any runtime issues and want to trace operations that take place with Java applications that utilize GPU, check the “GPU problem determination” subsection of the problem determination section of the “Linux User Guide for IBM SDK, Java Technology Edition” at the following website:

`http://www.ibm.com/support/knowledgecenter/SSYKE2`

Relying on the JVM logic to offload processing to a GPU

Certain Java functions are particularly appropriate for GPU processing. One class of such functions is array processing routines. For example, a data array can be sorted on a GPU faster than on a CPU. However, to benefit from this type of processing, the array must be of a sufficient size. This is required to justify the time of data movement between the CPU and the GPU.

JVM, which is shipped with IBM SDK, Java Technology Edition, is able to automatically offload certain Java functions. This happens when the JVM expects that the speed of data processing at the GPU outweighs the cost of data movement from main memory to GPU. This option allows you to leverage the GPU processing power without the need to change the source code. The ability to offload computations to a GPU is controlled by setting the system property when invoking JVM using the command line.

Example 29 lists a simple Java code that uses the *sort()* function from the *java.util.Arrays* package. This code runs without any changes on any platform that supports Java.

Example 29 Source file of a program that uses `sort()` function from `java.util.Arrays` package

```
import java.util.Arrays;
import java.util.Random;

public class BuiltInSort {
  public static void main(String[] args) {
    int N = 128*1024*1024;
    int[] toSort = new int[N];
    Random rnd = new Random();
    for (int i = 0; i < N; ++i) {
      toSort[i] = rnd.nextInt();
    }
    long startTime = System.nanoTime();
    Arrays.sort(toSort);
    long estimatedTime = System.nanoTime() - startTime;
    System.out.println("CPU\t" + N + '\t' + estimatedTime * 1e-9 + " seconds");
  }
}
```

Example 30 shows how to compile the code and run it by supplying various options to the JVM that is shipped with the IBM SDK, Java Technology Edition.

Example 30 Compiling and running the program that uses the `sort()` function (some output omitted)

```
$ javac BuiltInSort.java

$ java -Xmso4m -Xmx12g -Dcom.ibm.gpu.verbose  BuiltInSort
CPU     134217728       16.67377956 seconds

$ java -Xmso4m -Xmx12g -Dcom.ibm.gpu.verbose -Dcom.ibm.gpu.disable BuiltInSort
[IBM GPU]: [<...>]: System property com.ibm.gpu.disable=
[IBM GPU]: [<...>]: Disabling sort on the GPU
CPU     134217728       15.439728421000002 seconds

$ java -Xmso4m -Xmx12g -Dcom.ibm.gpu.verbose -Dcom.ibm.gpu.enable=sort BuiltInSort
[IBM GPU]: [<...>]: System property com.ibm.gpu.enable=sort
[IBM GPU]: [<...>]: Enabling sort on the GPU
[IBM GPU]: [<...>]: Discovered 2 devices
[IBM GPU]: [<...>]: Providing identifiers of discovered CUDA devices, found: 2
devices
[IBM GPU]: [<...>: Discovered devices have the following identifier(s):
0, 1
[IBM GPU]: [<...>]: Acquired device: 0
[IBM GPU]: [<...>]: Using device: 0 to sort int array of length: 134217728
[IBM GPU]: [<...>]: Sorted ints on device: 0 successfully
[IBM GPU]: [<...>]: Released device: 0
CPU     134217728       4.600549149 seconds
```

The *-Xmso* runtime option sets the initial stack size for operating system threads. The *-Xmx* option specifies the maximum size of the memory allocation pool (heap).

The system properties have the following meaning:

- *-Dcom.ibm.gpu.enable*. This system property controls the type of processing that can be offloaded by JVM to a GPU.
- *-Dcom.ibm.gpu.disable*. This option enables JVM to turn off the automatic GPU offloading capability.
- *-Dcom.ibm.gpu.verbose*. This option enables a detailed logging of how JVM handles GPU processing.

The code was run on an S824L system with 3.42 GHz cores and NVIDIA Tesla K40 GPU accelerators. The processing times seen in the output show that, by default, JVM does not offload the work to be performed by the *sort()* function to the GPU. When this feature is enabled, JVM discovers that the data processing requirements meet some specific threshold and offloads the sorting task to the GPU.

For more details about this feature, refer to *Enabling application processing on a graphics processing unit* in the section titled “Running Java applications of the Linux User Guide for IBM SDK, Java Technology Edition”, at the IBM Knowledge Center:

http://www.ibm.com/support/knowledgecenter/SSYKE2

The com.ibm.gpu application programming interface

You can use the *com.ibm.gpu* classes to explicitly specify in the code when you want to use GPU for certain tasks. Example 31 shows how to invoke a sorting routine that will run on a GPU. You can compile this code with the usual **`javac`** command, and execute the resulting Java byte code using ordinary **`java`** commands.

Example 31 Source file of a program that uses com.ibm.gpu classes

```
import java.util.Random;

import com.ibm.gpu.GPUConfigurationException;
import com.ibm.gpu.GPUSortException;
import com.ibm.gpu.Maths;

public class GpuSort {
  public static void main(String[] args) {
    int N = 128*1024*1024;
    int[] toSort = new int[N];
    Random rnd = new Random();
    for (int i = 0; i < N; ++i) {
      toSort[i] = rnd.nextInt();
    }
    long startTime = System.nanoTime();
    try {
      Maths.sortArray(toSort);
    } catch (GPUSortException e) {
      e.printStackTrace();
    } catch (GPUConfigurationException e) {
      e.printStackTrace();
    }
    long estimatedTime = System.nanoTime() - startTime;
    System.out.println("GPU\t" + N + '\t' + estimatedTime * 1e-9 + " seconds");
  }
}
```

Figure 8 shows how the performance of a sorting function built into Java compares to the performance of a sorting function done in a GPU. Not only is the performance of the GPU sorting function higher, but the routine is more scalable than a Java native sorting routine.

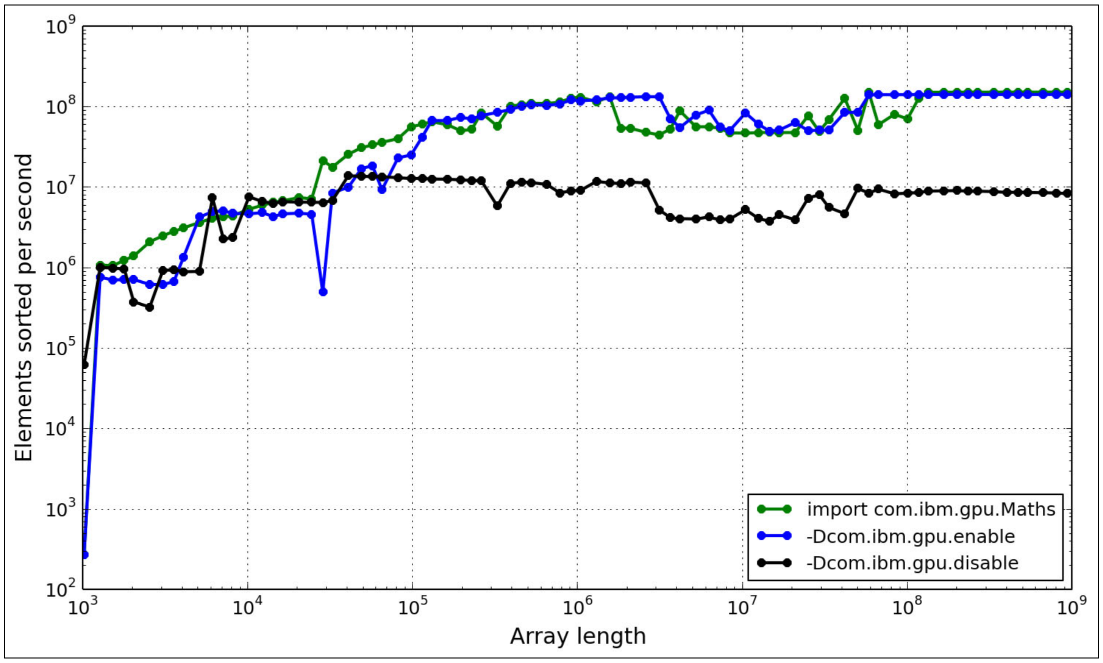


Figure 8 The performance of the Java sort() function executed on CPU and GPU

In addition, with the *com.ibm.gpu* classes, you can enable verbose output programmatically. You can also provide a device identifier with the function arguments, if you want to target a specific GPU device.

For more information about the *com.ibm.gpu* classes, see *The ibm.com.gpu application programming interface in the Linux User Guide for IBM SDK, Java Technology Edition*, at:

http://www.ibm.com/support/knowledgecenter/SSYKE2

The CUDA4J application programming interface

You can improve the performance of your Java applications by offloading certain processing functions from a CPU to a GPU. So far, this publication has only described the functions that are supplied as part of IBM JDK packages.

CUDA4J is an application programming interface (API) that is offered by IBM for developing applications in which you can specify exactly when and how to use the GPU for application processing.

The CUDA4J API enables you to develop applications that can move data between the Java heap and memory buffers on the GPU. On the GPU, kernels that are written in the C programming language can be invoked to process that data, and the results can be moved back into the Java heap under the control of your Java application. The CUDA4J API provides classes for setting up buffers that allow data transfer between Java memory and device memory.

For information about com.ibm.cuda classes that supply the CUDA4J API, see “CUDA4J” in the Linux User Guide for IBM SDK, Java Technology Edition, at the following site:

http://www.ibm.com/support/knowledgecenter/SSYKE2

Nsight

The Nsight Eclipse Edition is a CUDA development environment provided by NVIDIA as part of the CUDA toolkit's cross-platform development packages. Nsight Eclipse Edition 6.0.0 is included with CUDA 5.5 for POWER8, and is built on Eclipse Platform 3.8.1 and CDT (C/C++ Development Tools). It includes added-value specific features for easing the GPU development cycle of coding, building, launching, debugging, and analyzing.

Note: Nsight runs on a x86-64 computer, not on the target POWER8 system.

This section explains the basics of creating, editing, building, and launching projects from within the environment. More details about Nsight for Eclipse Edition can be found at:

http://developer.nvidia.com/nsight-eclipse-edition

Note: You might notice differences in version numbers between the CUDA 5.5 toolkit and some components (for example, Nsight, CUDA debugger, and so on) in screen captures and examples. This is mostly addressed in the CUDA 5.5-54 release, and should not be present in CUDA 7.0.

After the CUDA cross-platform development packages have been installed, and the environment variables configured (see "Software stack" on page 14) on your personal development computer, Nsight can be opened with the **`nsight`** command shown in Example 32.

Example 32 Start Nsight development environment

```
$ nsight &
```

Create projects

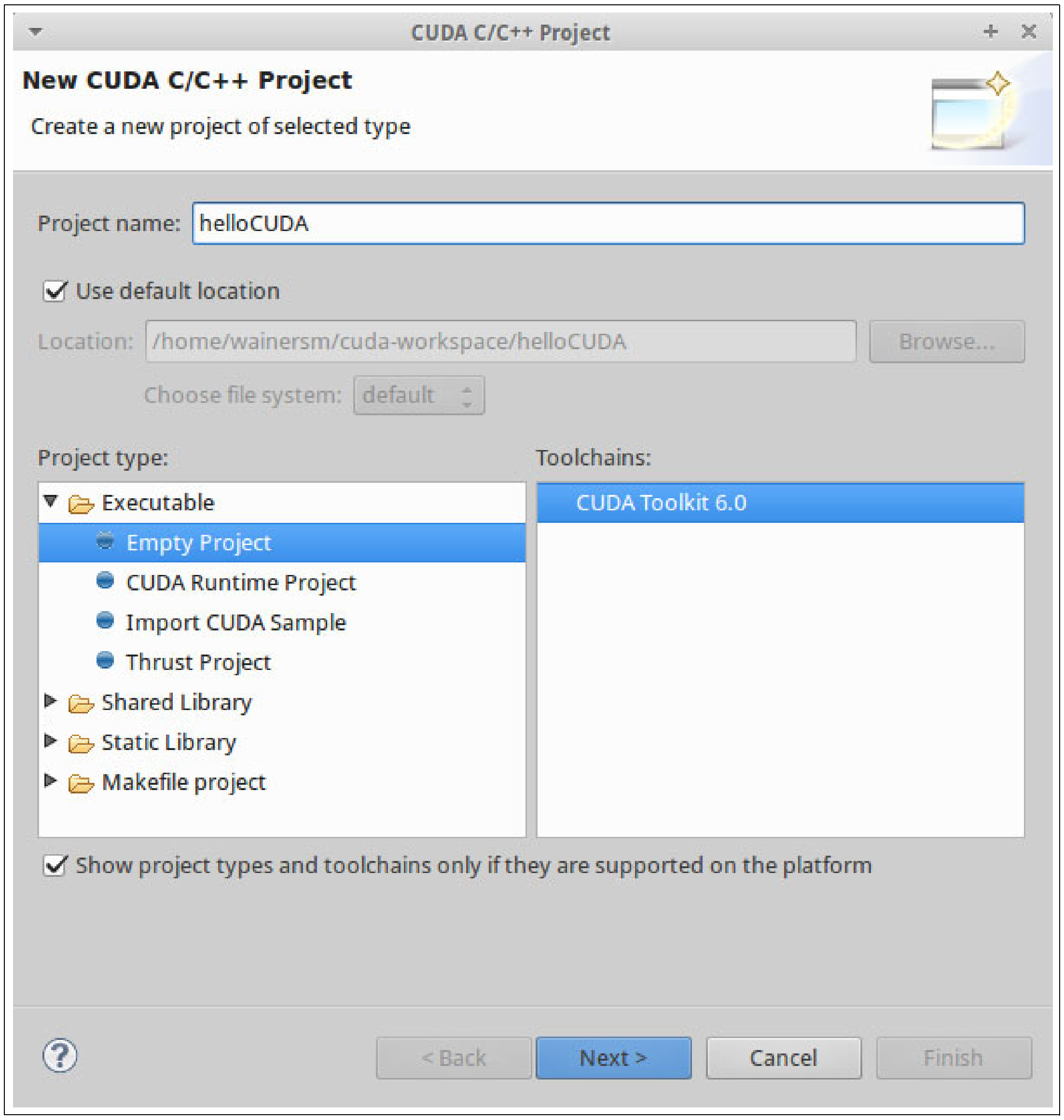
Nsight allows you to create and develop projects that are either local (with or without cross-compiling) or remote. However, the suggested mode in this paper is to develop remotely. The environment generates a synchronized project where remote source files are mirrored locally so that the code editor does not show interruptions, due to network latency, when you are typing. To enable synchronization operations, you must install and configure the Git versioning tool in the remote machine, as shown in Example 33. You must also install the CUDA toolkit for POWER8 in the remote target machine (see "Software stack" on page 14).

Example 33 Install and configure Git in Ubuntu

```
$ sudo apt-get install git
$ git config --global user.name "Wainer dos Santos Moschetta"
$ git config --global user.email "wainersm@br.ibm.com"
```

Follow these steps to create a remote synchronized project:

1. At the menu bar, click **File** → **New** → **CUDA C/C++ Project**. The New Project window appears, as shown in Figure 9 on page 45.


CUDA C/C++ Project

New CUDA C/C++ Project

Create a new project of selected type

Project name: helloCUDA

☑ Use default location

Location: /home/wainersm/cuda-workspace/helloCUDA

Choose file system: default

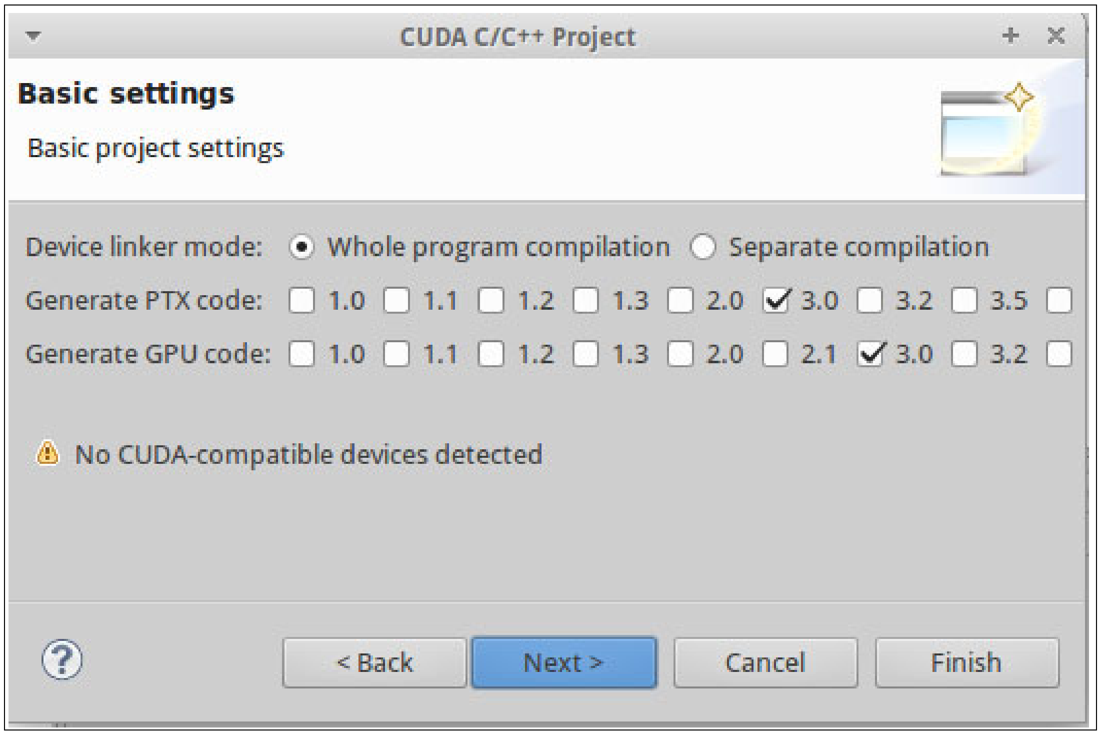
Project type:
- Executable
  - Empty Project
  - CUDA Runtime Project
  - Import CUDA Sample
  - Thrust Project
- Shared Library
- Static Library
- Makefile project

Toolchains:
- CUDA Toolkit 6.0

☑ Show project types and toolchains only if they are supported on the platform

Figure 9 Nsight new CUDA C/C++ project window

2. In the Project type area (left side of panel), select **Executable** → **Empty Project** to create an Eclipse managed makefile project, then click **Next**. The “Basic settings” panel appears, as shown in Figure 10 on page 46. In the Basic settings panel, select the device linker mode that should be taken at compilation time, and also the compatibility levels of the generated executable. See “Compilers” on page 36 for available options for GPU on POWER8. These options can be changed later on in the project properties window.



Basic settings

Basic project settings

Device linker mode: [✓ Whole program compilation] Separate compilation

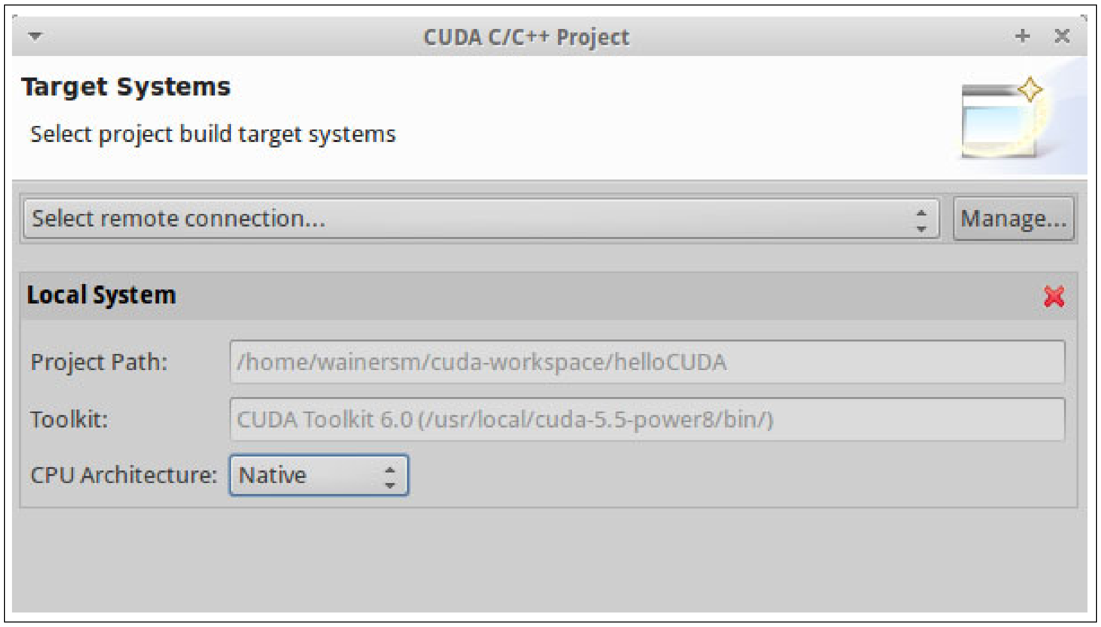
Generate PTX code: ☐ 1.0 ☐ 1.1 ☐ 1.2 ☐ 1.3 ☐ 2.0 ✓ 3.0 ☐ 3.2 ☐ 3.5 ☐

Generate GPU code: ☐ 1.0 ☐ 1.1 ☐ 1.2 ☐ 1.3 ☐ 2.0 ☐ 2.1 ✓ 3.0 ☐ 3.2 ☐

No CUDA-compatible devices detected

Figure 10 Basic settings at the new CUDA C/C++ project

3. Click **Next** to open the Target Systems window as shown in Figure 11. If a connection to the compute machine exists, select an option in the drop-down list. Otherwise, click **Manage** to add a connection to the machine. The Remote Connections window opens (see Figure 12 on page 47), then click **Finish**.



Target Systems

Select project build target systems

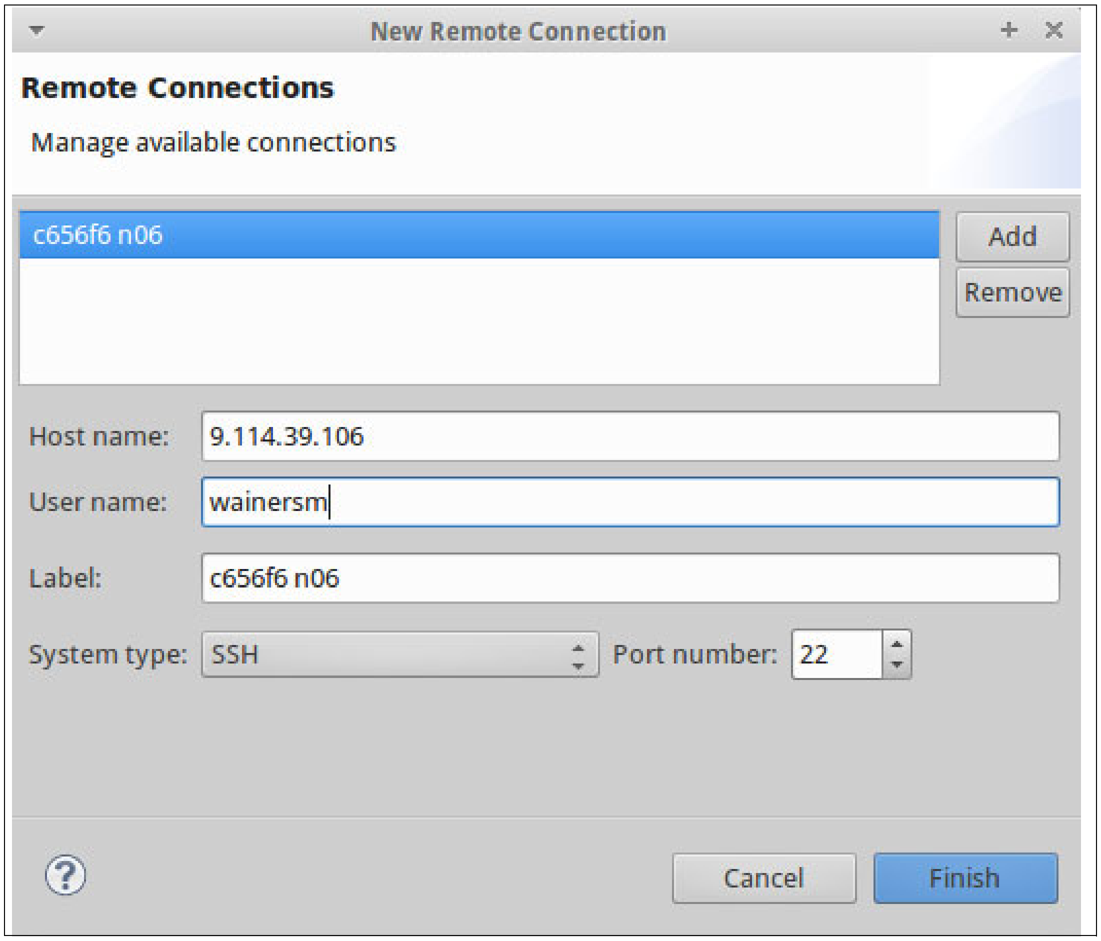
Local System

Project Path: /home/wainersm/cuda-workspace/helloCUDA

Toolkit: CUDA Toolkit 6.0 (/usr/local/cuda-5.5-power8/bin/)

CPU Architecture: Native

Figure 11 Initial target systems at the new CUDA C/C++ project



Remote Connections

Manage available connections

c656f6 n06

Host name: 9.114.39.106

User name: wainersm

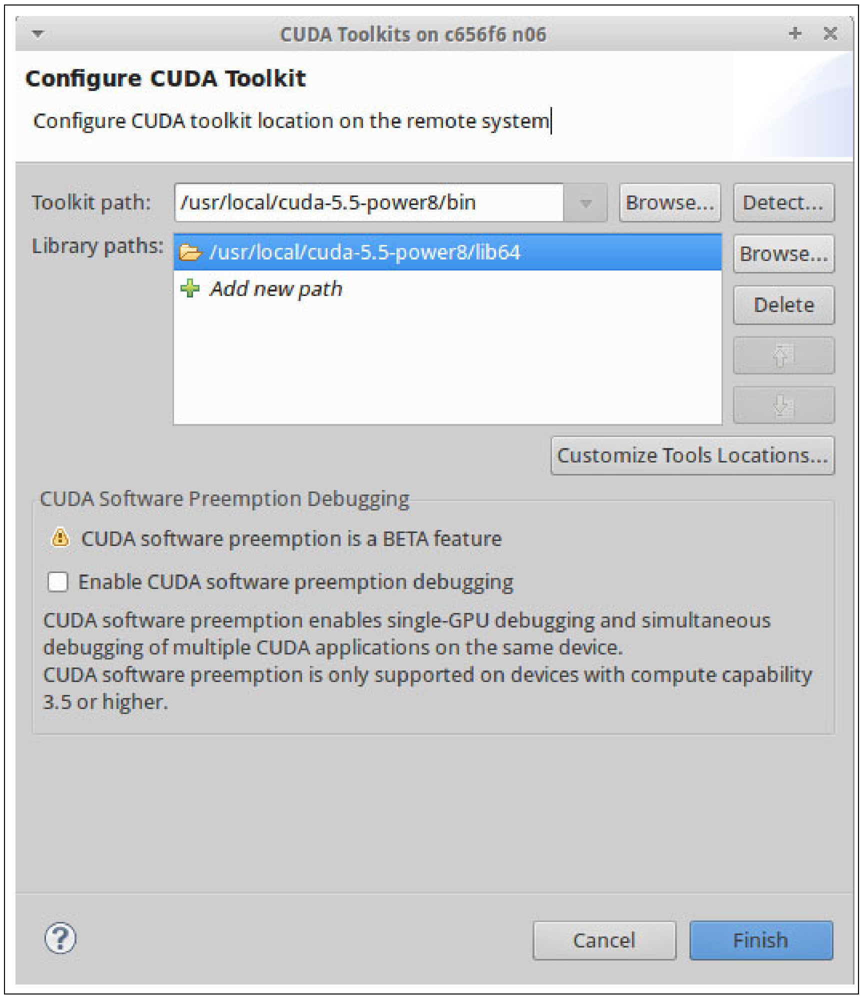
Label: c656f6 n06

System type: SSH

Port number: 22

Figure 12 New connection configuration window

4. The next step is to configure the location of the CUDA toolkit and libraries in the remote machine. Click **Manage** at the connection box, and the menu as shown in Figure 13 on page 48 appears. Set the location of the CUDA toolkit (*<cuda_installation_dir>/bin/*) and libraries (*<cuda_installation_dir>/lib64/*), then click **Finish**.


CUDA Toolkits on c656f6 n06

Configure CUDA Toolkit

Configure CUDA toolkit location on the remote system

Toolkit path: /usr/local/cuda-5.5-power8/bin

Library paths: /usr/local/cuda-5.5-power8/lib64

Add new path

CUDA Software Preemption Debugging

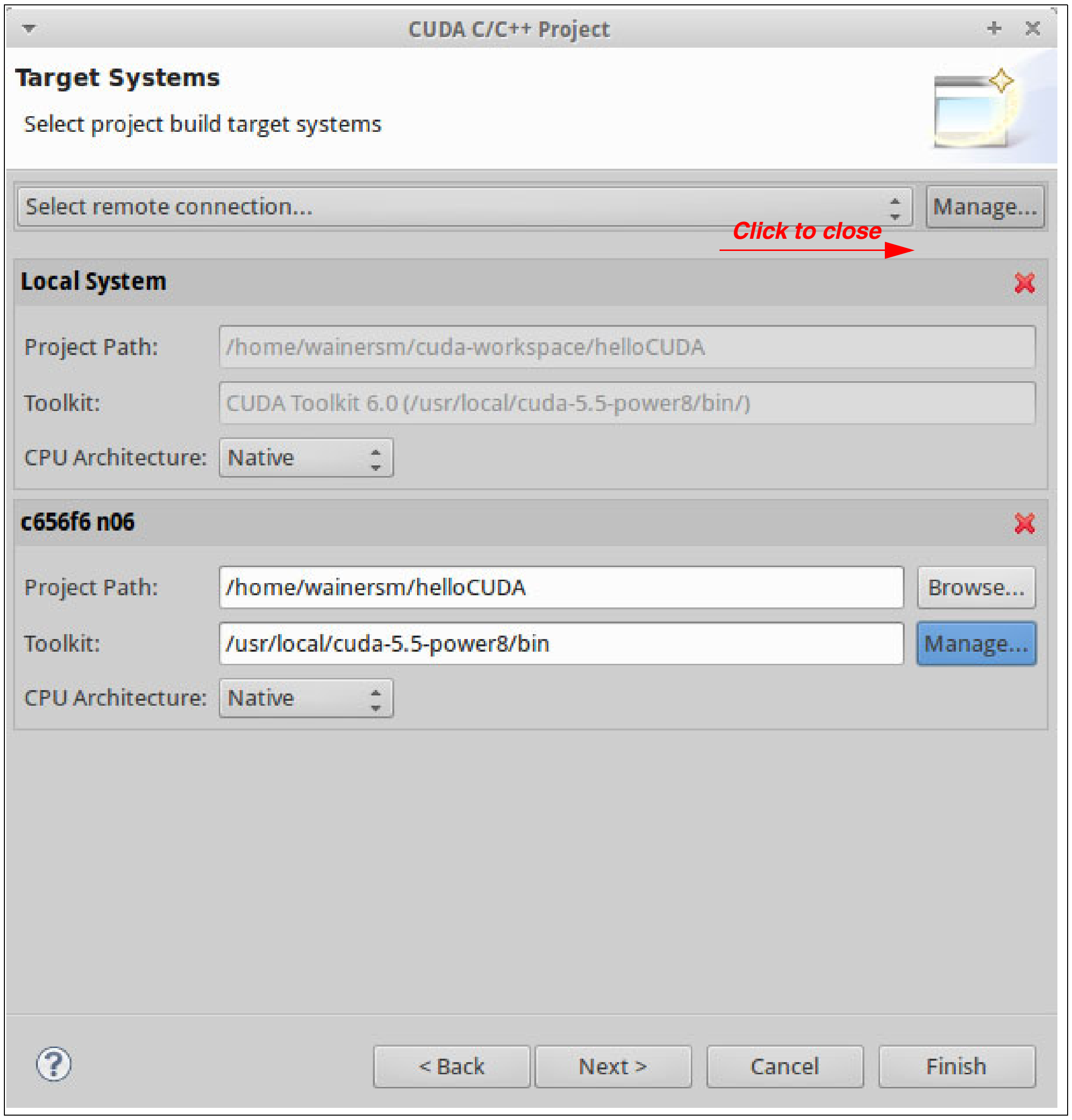
CUDA software preemption is a BETA feature

☐ Enable CUDA software preemption debugging

CUDA software preemption enables single-GPU debugging and simultaneous debugging of multiple CUDA applications on the same device.
CUDA software preemption is only supported on devices with compute capability 3.5 or higher.

Figure 13 Setting the CUDA toolkit and libraries path in the remote machine

5. Until now, the remote connection has been set properly along with a local one that exists by default. To avoid triggering builds locally, delete the local configuration as shown with the red arrow in Figure 14 on page 49.


CUDA C/C++ Project

Target Systems

Select project build target systems

Select remote connection...

Local System

Project Path: /home/wainersm/cuda-workspace/helloCUDA

Toolkit: CUDA Toolkit 6.0 (/usr/local/cuda-5.5-power8/bin/)

CPU Architecture: Native

c656f6 n06

Project Path: /home/wainersm/helloCUDA

Toolkit: /usr/local/cuda-5.5-power8/bin

CPU Architecture: Native

Figure 14 Target systems at the new CUDA C/C++ project window

6. Click **Finish** to generate the project in the Eclipse workspace.

The result of the preceding steps is a project that has a development cycle of build, run, and debug in the remote GPU enabled machine.

To create a sample application, click **File** → **New** → **Source file**, and choose the CUDA C Bitreverse Application template.

Building projects

The Eclipse CDT (C/C++ Development Tools) has full control over the build process of the project created in Figure 14. In simple terms, it generates several Makefiles from the project settings to build the source code on the remote side. It is possible to generate other kinds of projects, for instance, makefile or shell-script based, where the developer is in control of the construction process, but they are not covered in this publication.

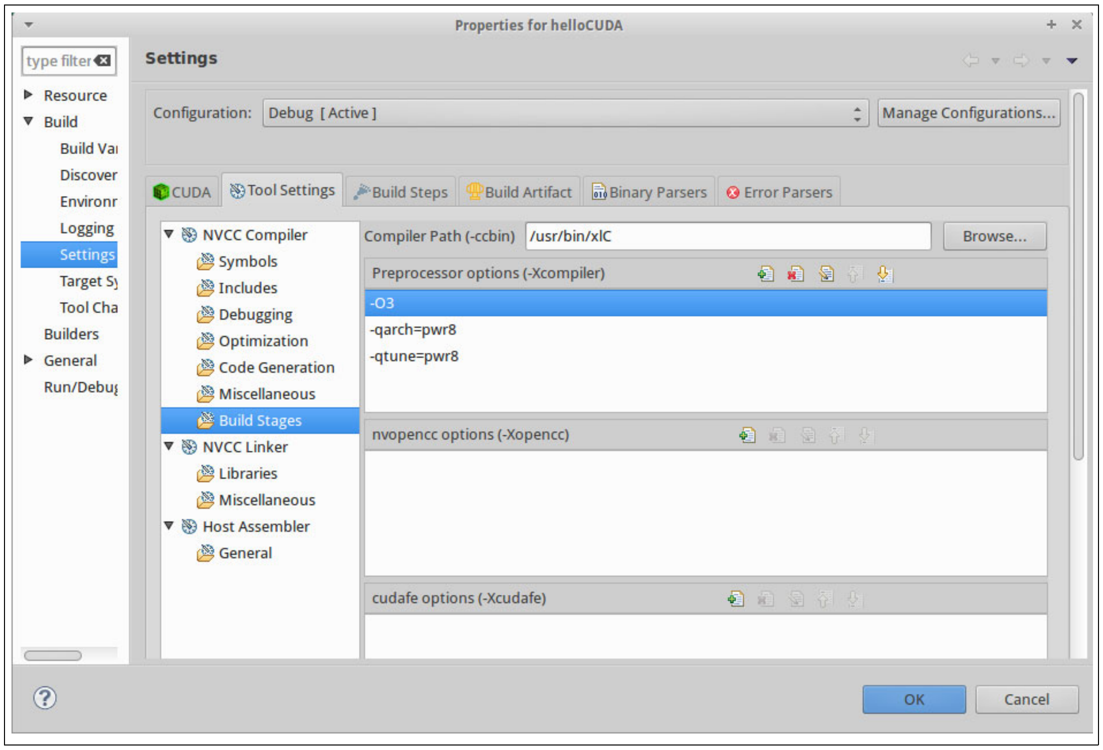
To build the project, either click the hammer icon in the main toolbar, or select **Project** → **Build Project**.

Nsight uses preset options of the nvcc compiler from a location that was defined in the remote connection setup (see Figure 14 on page 49), and it chooses the gcc or the g++ compiler, which are in the system PATH variable.

There are several properties in the project build settings that change the default behavior. Click **Project** → **Properties** on the main toolbar, unroll **Build** on the left side panel, and click **Settings** to see most of the properties.

To change the back-end compiler, do the following:

1. From the build settings window, switch to the **Tool Settings** tab at the right side panel.
2. On the left side panel, expand the **NVCC Compiler** and click **Build Stages**. Some of the properties for controlling the nvcc build stages are shown in Figure 15.



Configuration: Debug [Active]

Compiler Path (-ccbin) /usr/bin/xlC

Preprocessor options (-Xcompiler)

-O3
-qarch=pwr8
-qtune=pwr8

nvopencc options (-Xopencc)

cudafe options (-Xcudafe)

Figure 15   Building settings for NVCC stages

3. Enter the path to the compiler backend in the Compiler Path (-ccbin) field. Click **OK**.

Figure 15 shows the configuration for setting the `xlC` back-end compiler, as well as for passing flags **`-O3`**, **`-qarch=pwr8`**, and **`-qtune=pwr8`** (see the **Preprocessor** options field in the figure).

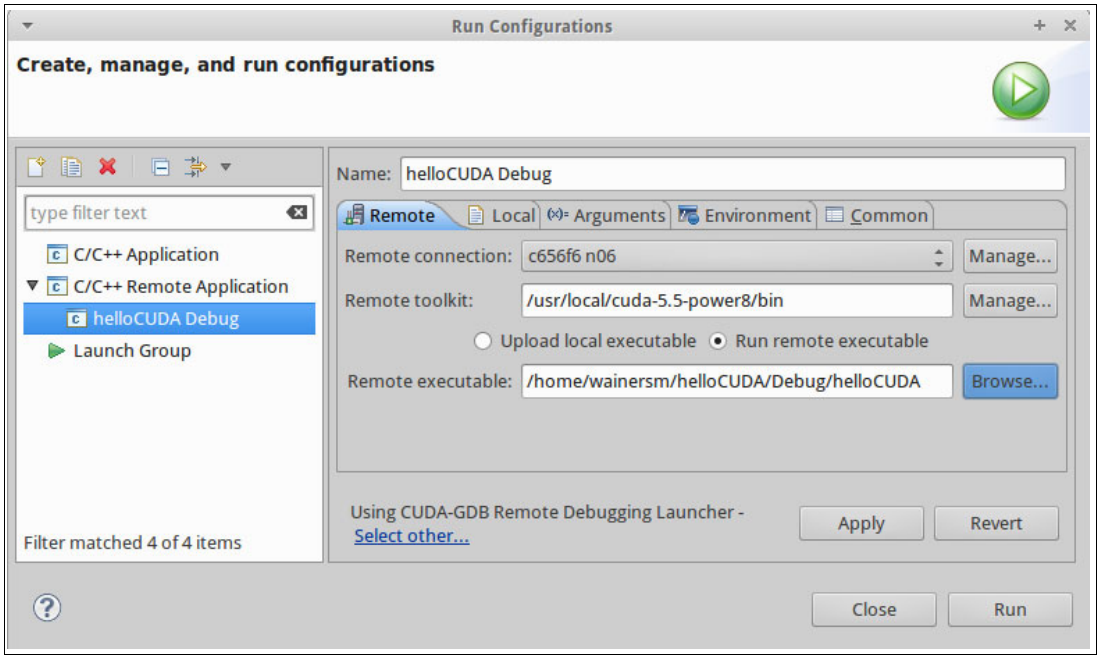
To change the CUDA runtime library linkage, proceed as follows:

1. From the build settings window, switch to the **Tool Settings** tab on the right side panel.
2. On the left side panel, expand the **NVCC Linker** and click **Libraries**.
3. Set the **CUDA Runtime Library** drop-box list to either **None**, **Shared**, or **Static**. The default value is **Static**.

Launching applications

The project configuration that was previously set allows you to run the CUDA application in the remote GPU-enabled machine. To run the application, proceed as follows:

1. Click **Run** → **Run Configurations** on the main toolbar.
2. Double-click **C/C++ Remote Application**. The Run Configurations window appears, as shown in Figure 16.


Run Configurations

Create, manage, and run configurations

Name: helloCUDA Debug

Remote connection: c656f6 n06

Remote toolkit: /usr/local/cuda-5.5-power8/bin

○ Upload local executable ● Run remote executable

Remote executable: /home/wainersm/helloCUDA/Debug/helloCUDA

Using CUDA-GDB Remote Debugging Launcher - Select other...

Filter matched 4 of 4 items

Figure 16 Run Configurations window

3. On the **Remote** tab, change the **Remote connection** drop-down menu from **Local** to the label of the connection.
4. Enable the **Run remote executable** radio box option.
5. Click **Browse** to select the path to the executable at the remote machine. Use the **Arguments** and **Environment** tabs to set any application-specific properties.
6. Click **Run**.

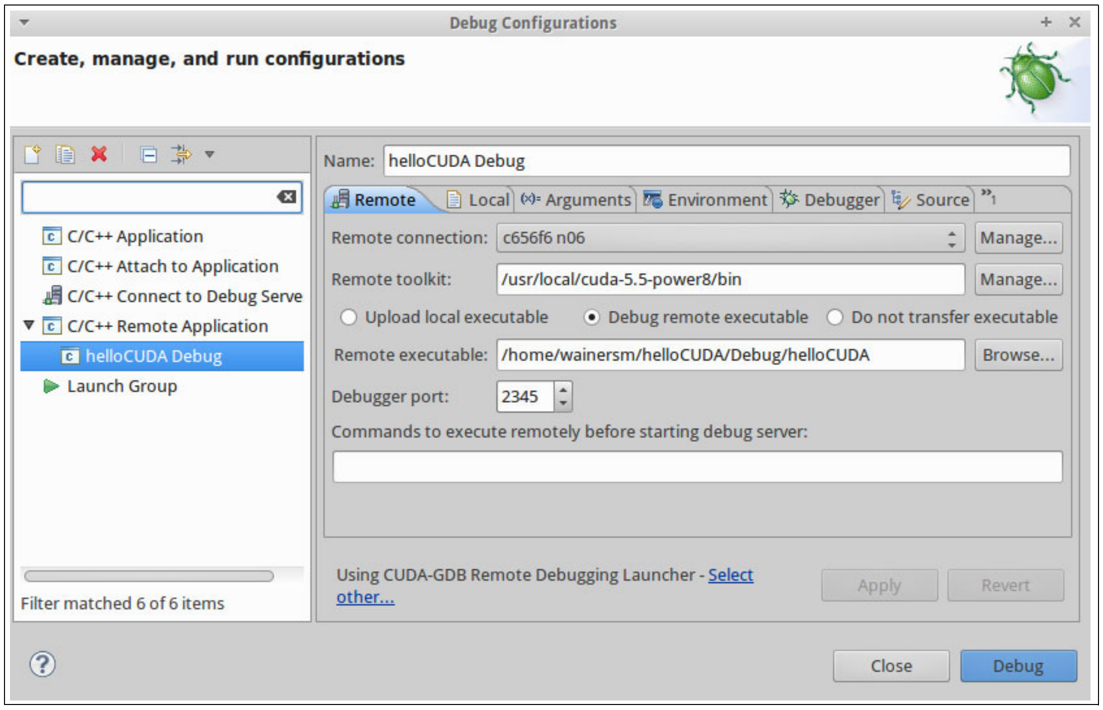
The executable output is shown in the Eclipse console on the bottom panel.

Debug applications

You can also find problems in the CUDA executables from within the Eclipse environment. Nsight provides a graphical debugger that starts the `cuda-gdbserver` at the remote machine, then connects in, which allows you to start a debug session. Nsight is also able to connect to an already running `cuda-gdbserver`, but this topic is not covered in this publication.

To begin a debug session, follow these steps:

1. Click **Run** → **Debug Configurations** in the main toolbar.
2. Double-click **C/C++ Remote Application**. This window is shown in Figure 17 on page 52. The **Remote** tab can be used to specify connection details.



Name: helloCUDA Debug

Remote connection: c656f6 n06

Remote toolkit: /usr/local/cuda-5.5-power8/bin

( ) Upload local executable [✓ Debug remote executable] ( ) Do not transfer executable

Remote executable: /home/wainersm/helloCUDA/Debug/helloCUDA

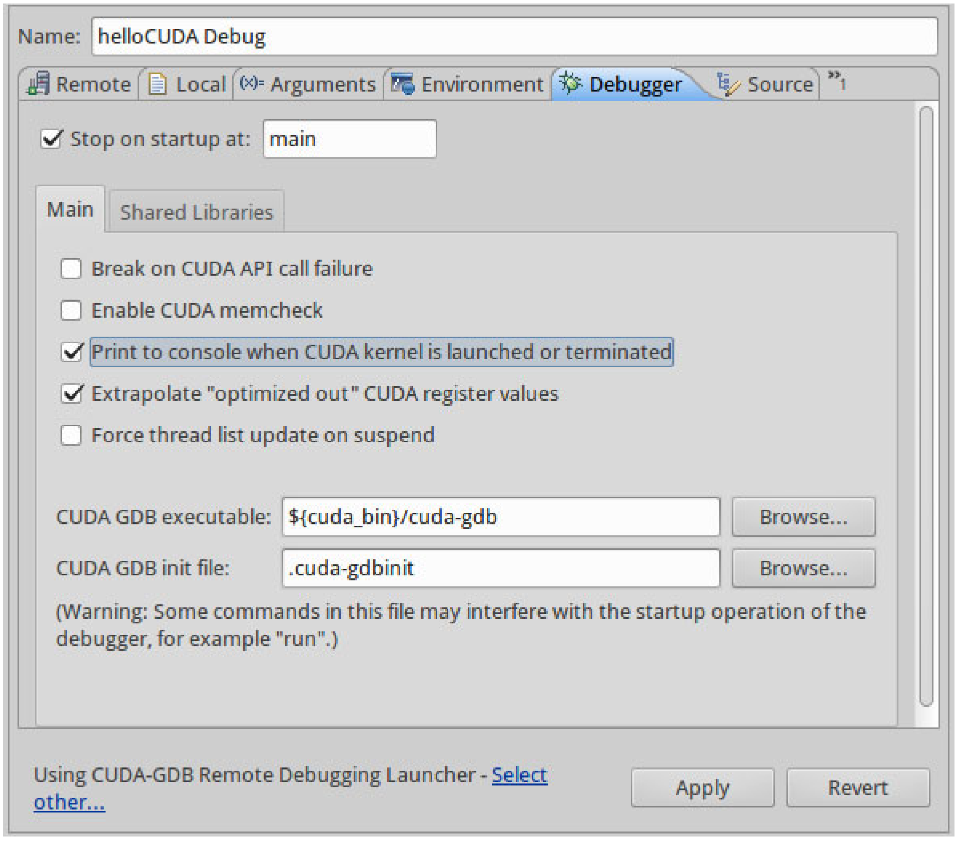
Debugger port: 2345

Commands to execute remotely before starting debug server:

Using CUDA-GDB Remote Debugging Launcher - Select other...

Figure 17 Remote tab at the debug configuration window

3. Switch to the **Debugger** tab and complete more configuration details for the session. By default, it halts at the main method but also in any breakpoint set in the host and kernel codes. Figure 18 on page 53 is an example of a default debugger configuration.


Name: helloCUDA Debug

[✓] Stop on startup at: main

[ ] Break on CUDA API call failure
[ ] Enable CUDA memcheck
[✓] Print to console when CUDA kernel is launched or terminated
[✓] Extrapolate "optimized out" CUDA register values
[ ] Force thread list update on suspend

CUDA GDB executable: ${cuda_bin}/cuda-gdb
CUDA GDB init file: .cuda-gdbinit
(Warning: Some commands in this file may interfere with the startup operation of the debugger, for example "run".)

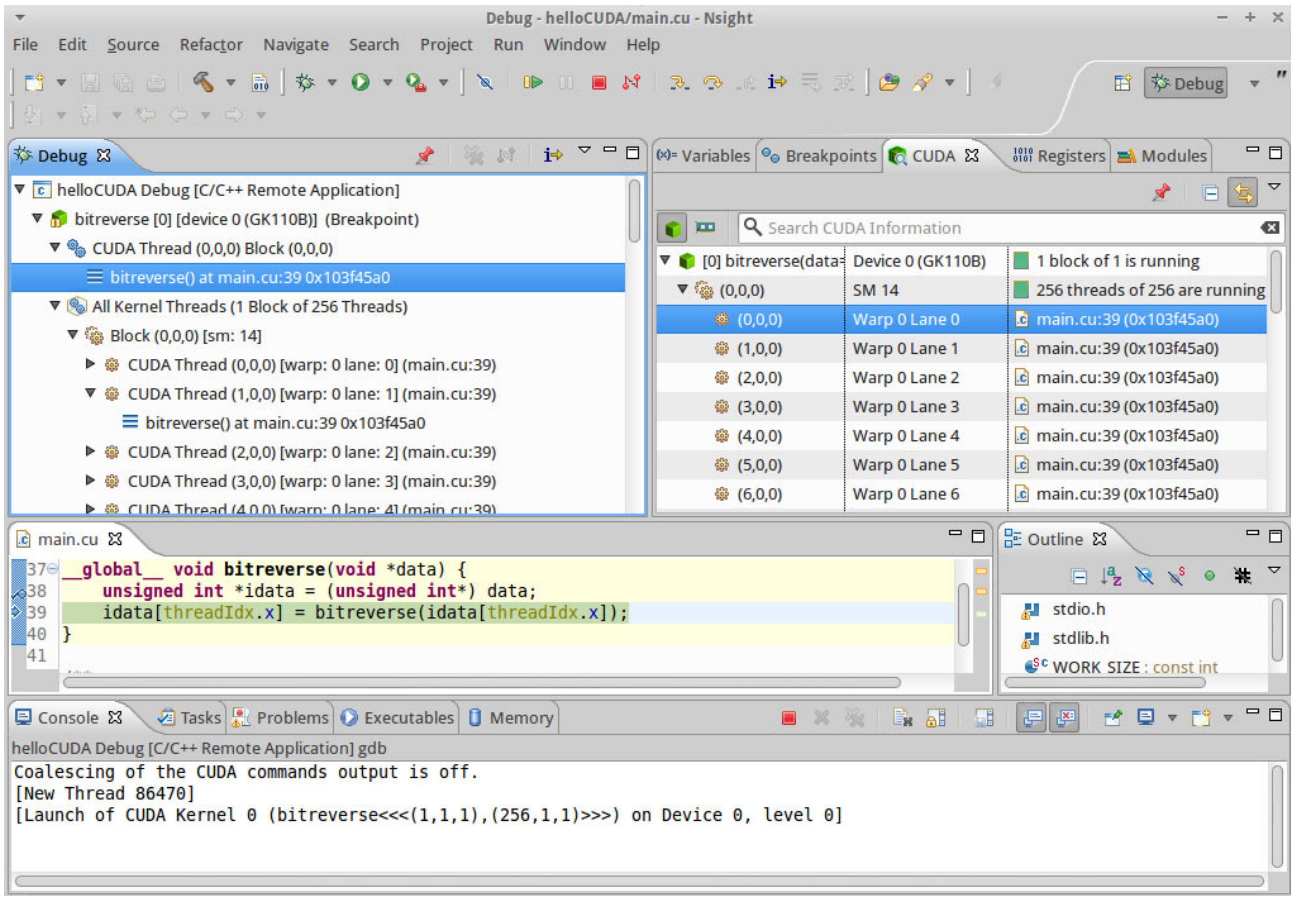
Using CUDA-GDB Remote Debugging Launcher - Select other...

Figure 18 Remote tab at Debug configuration window

4. Use the **Arguments** and the **Environment** tabs to set any application-specific properties. Click **Debug** to begin the session.

After a connection to the remote **cuda-gdbserver** has been established, the Eclipse debugger opens. The Eclipse debugger provides information about the host and kernel source codes and allows you to debug them. Figure 19 on page 54 shows the following important debug information:

- Number of threads
- Threads per block and grid
- Threads per warp and lanes within GPU device
- Breakpoints along with source code
- GDB console output


Debug - helloCUDA/main.cu - Nsight
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Debug
Variables Breakpoints CUDA Registers Modules
Search CUDA Information
main.cu
Outline
stdio.h
stdlib.h
WORK_SIZE : const int
Console Tasks Problems Executables Memory

```
helloCUDA Debug [C/C++ Remote Application]
  bitreverse [0] [device 0 (GK110B)] (Breakpoint)
    CUDA Thread (0,0,0) Block (0,0,0)
      bitreverse() at main.cu:39 0x103f45a0
    All Kernel Threads (1 Block of 256 Threads)
      Block (0,0,0) [sm: 14]
        CUDA Thread (0,0,0) [warp: 0 lane: 0] (main.cu:39)
        CUDA Thread (1,0,0) [warp: 0 lane: 1] (main.cu:39)
          bitreverse() at main.cu:39 0x103f45a0
        CUDA Thread (2,0,0) [warp: 0 lane: 2] (main.cu:39)
        CUDA Thread (3,0,0) [warp: 0 lane: 3] (main.cu:39)
        CUDA Thread (4,0,0) [warp: 0 lane: 4] (main.cu:39)
```

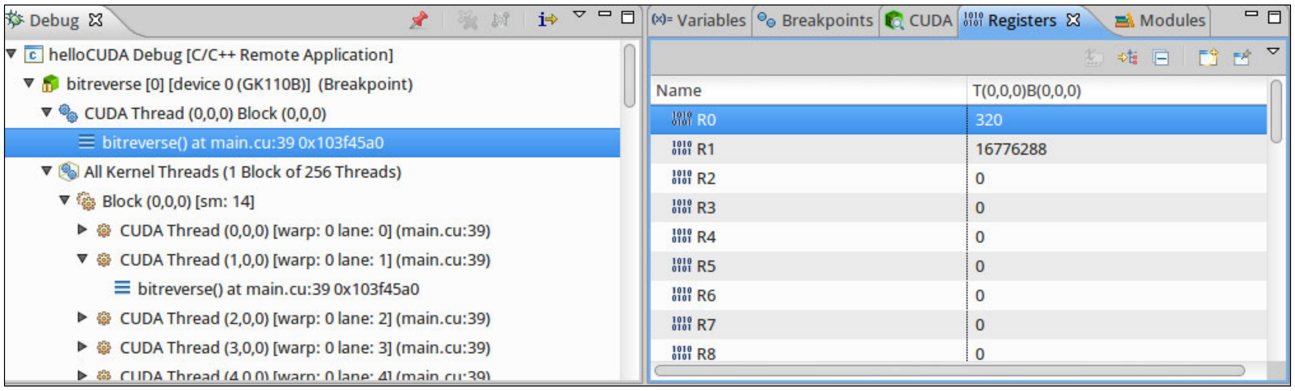
[0] bitreverse(data=	Device 0 (GK110B)	🟩 1 block of 1 is running
(0,0,0)	SM 14	🟩 256 threads of 256 are running
(0,0,0)	Warp 0 Lane 0	main.cu:39 (0x103f45a0)
(1,0,0)	Warp 0 Lane 1	main.cu:39 (0x103f45a0)
(2,0,0)	Warp 0 Lane 2	main.cu:39 (0x103f45a0)
(3,0,0)	Warp 0 Lane 3	main.cu:39 (0x103f45a0)
(4,0,0)	Warp 0 Lane 4	main.cu:39 (0x103f45a0)
(5,0,0)	Warp 0 Lane 5	main.cu:39 (0x103f45a0)
(6,0,0)	Warp 0 Lane 6	main.cu:39 (0x103f45a0)

```
__global__ void bitreverse(void *data) {
    unsigned int *idata = (unsigned int*) data;
    idata[threadIdx.x] = bitreverse(idata[threadIdx.x]);
}
```

```
helloCUDA Debug [C/C++ Remote Application] gdb
Coalescing of the CUDA commands output is off.
[New Thread 86470]
[Launch of CUDA Kernel 0 (bitreverse<<<(1,1,1),(256,1,1)>>>) on Device 0, level 0]
```

Figure 19 Debugger main view

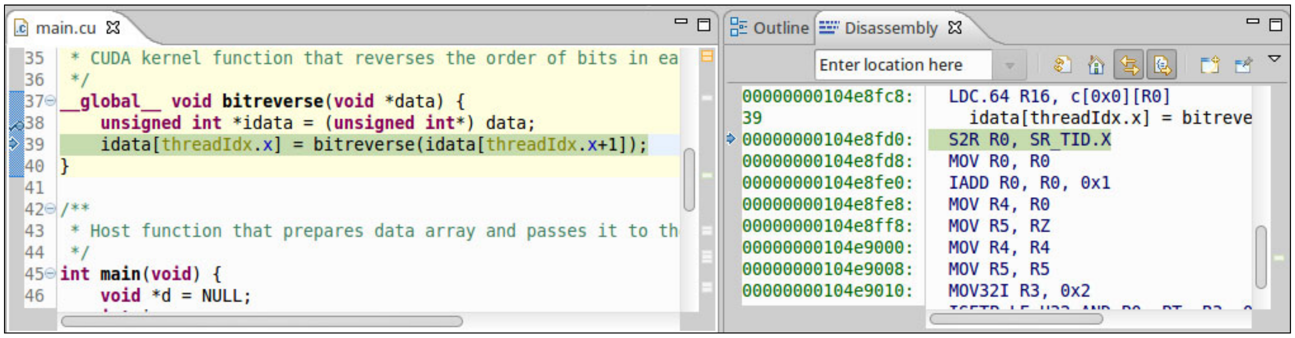
It is possible to see the GPU registers value by switching to **Registers** on the right side frame, as shown in Figure 20. There is also a view for disassembling code, as shown in Figure 21 on page 55.



```
helloCUDA Debug [C/C++ Remote Application]
  bitreverse [0] [device 0 (GK110B)] (Breakpoint)
    CUDA Thread (0,0,0) Block (0,0,0)
      bitreverse() at main.cu:39 0x103f45a0
    All Kernel Threads (1 Block of 256 Threads)
      Block (0,0,0) [sm: 14]
        CUDA Thread (0,0,0) [warp: 0 lane: 0] (main.cu:39)
        CUDA Thread (1,0,0) [warp: 0 lane: 1] (main.cu:39)
          bitreverse() at main.cu:39 0x103f45a0
        CUDA Thread (2,0,0) [warp: 0 lane: 2] (main.cu:39)
        CUDA Thread (3,0,0) [warp: 0 lane: 3] (main.cu:39)
        CUDA Thread (4,0,0) [warp: 0 lane: 4] (main.cu:39)
```

Name	T(0,0,0)B(0,0,0)
R0	320
R1	16776288
R2	0
R3	0
R4	0
R5	0
R6	0
R7	0
R8	0

Figure 20 GPU registers tab at debug main view



```
 * CUDA kernel function that reverses the order of bits in ea
 */
__global__ void bitreverse(void *data) {
    unsigned int *idata = (unsigned int*) data;
    idata[threadIdx.x] = bitreverse(idata[threadIdx.x+1]);
}

/**
 * Host function that prepares data array and passes it to th
 */
int main(void) {
    void *d = NULL;
```

```
00000000104e8fc8:   LDC.64 R16, c[0x0][R0]
39                    idata[threadIdx.x] = bitreve
00000000104e8fd0:   S2R R0, SR_TID.X
00000000104e8fd8:   MOV R0, R0
00000000104e8fe0:   IADD R0, R0, 0x1
00000000104e8fe8:   MOV R4, R0
00000000104e8ff0:   MOV R5, RZ
00000000104e9000:   MOV R4, R4
00000000104e9008:   MOV R5, R5
00000000104e9010:   MOV32I R3, 0x2
```

Figure 21   Code disassembly tab at debug main view

NVIDIA CUDA Toolkit code samples

The NVIDIA CUDA toolkit ships with a set of code samples that cover many of the CUDA features (see “CUDA paradigm” on page 36) and GPU accelerated libraries (see “GPU accelerated libraries” on page 39).

To get started with the examples, copy them to a directory to which you have write access. For example:

```
$ cp -r <cuda_installation_dir>/samples/ $HOME/cuda-samples
```

To build an individual example, run the **make** command in its respective directory. To build all the examples at once, execute **make** in the top-level directory that contains the code samples.

By default, the samples are compiled with the **g++** compiler. To enforce the use of the IBM XL C/C++ compiler, explicitly set the value for the *GCC* variable:

```
$ make GCC=xlC
```

You can clean the object directories and binary files by running:

```
$ make clean
```

Tuning and debugging

A CUDA program is essentially interleaving portions of execution code that at some point run on a CPU (host), at other points in a GPU (device), while concomitant to each other.

Running on different computational hardware (host versus device) means that CUDA execution environments must be understood before any optimization efforts because they provide distinct purpose and computational resources (threading model, memory capabilities, and so on) with notable influence on performance. Consult “NVIDIA Tesla K40 GPU” on page 10 for an overview of GPU hardware. Regarding application optimizations for POWER8, extensive documentation is available and the most essential can be found in *Performance Optimization and Tuning Techniques for IBM Processors, including IBM POWER8*, SG24-8171.

Heterogeneous parallel programming paradigms like CUDA offer many possible scenarios for the application. The characterization of the application is important for determining techniques and strategies to be employed. For example, you might have an application that:

- Runs completely in the CPU side. No GPU operation is in use.

- Runs GPU portions only by using CUDA-accelerated third-party libraries. No internal algorithm using CUDA kernels are executed.
- Runs heavily in the CPU but some computations are GPU offloaded to speed them up.
- Runs almost entirely in the GPU. The host CPU is only used for data feeding and solution consolidation.

Therefore, understanding the application behavior in run time is often the first step toward performance optimization. Here, profiling and tracing tools play fundamental roles because they are handy for application characterization, are used to show hotspots and bottlenecks, and are also used to discover inefficiencies in the use of computational resources. This introduces the main profiling tools for CUDA C/C++.

The broad use of performance-driven programming techniques and strategies can be employed to take advantage of GPU as much as possible. With this in mind, it is suggested that you read the “CUDA C Best Practices Guide”, which is available at the following site:

http://docs.nvidia.com/cuda/cuda-c-best-practices-guide

CUDA profiling tools

The CUDA 5.5 toolkit for POWER8 includes the **nvprof** command line profiling tool. **nvprof** can be used to:

- Identify performance hotspots
- Analyze behavior in areas of potential performance improvement

By default, **nvprof** shows the total time spent in each application kernel, the number of calls, and the average time per call.

Example 34 shows the output of **nvprof** for the *quasirandomGenerator* application (see “NVIDIA CUDA Toolkit code samples” on page 55). The application spends about 35% of the time copying data from the device to the host and 64% in kernel execution (*quasirandomGeneratorKernel* is responsible for about 48% of this number).

Example 34 nvprof. Identify hotspots in CUDA application

```
$ nvprof ./4_Finance/quasirandomGenerator/quasirandomGenerator
./4_Finance/quasirandomGenerator/quasirandomGenerator Starting...

==8071== NVPROF is profiling process 8071, command:
./4_Finance/quasirandomGenerator/quasirandomGenerator
GPU Device 0: "Tesla K40m" with compute capability 3.5

Allocating GPU memory...
Allocating CPU memory...

<... output ommitted ...>

Shutting down...
==8071== Profiling application: ./4_Finance/quasirandomGenerator/quasirandomGenerator
==8071== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 47.89%  9.1456ms        21  435.51us  431.30us  438.27us  quasirandomGeneratorKernel(float*,
unsigned int, unsigned int)
 34.89%  6.6623ms         2  3.3312ms  3.3154ms  3.3469ms  [CUDA memcpy DtoH]
 16.55%  3.1612ms        21  150.53us  147.74us  151.97us  inverseCNDKernel(float*, unsigned
int*, unsigned int)
  0.66%  126.59us         2  63.296us  62.816us  63.776us  [CUDA memset]
  0.01%  1.6320us         1  1.6320us  1.6320us  1.6320us  [CUDA memcpy HtoD]
```

The tool is able to generate predefined metrics about GPU hardware usage, including occupancy, utility and throughput. These are generated from hardware events data that is collected while the CUDA application runs. Use *--query-events* and *--query-metrics* **nvprof** options to list all events and metrics that are available for profiling, respectively.

A good starting point is to check the overall GPU's capacity usage by the application. For example, the achieved_occupancy metric could be used. See Table 3.

All utilization metrics can be used together to find a hotspot in the hardware subsystems that the application mostly uses. Use the metrics in Table 3 to understand the application's behavior from a GPU internals perspective.

Table 3 nvprof. utilization metrics

Metric	Description
achieved_occupancy	Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor.
l1_shared_utilization	The utilization level of the L1/shared memory relative to peak utilization.
l2_utilization	The utilization level of the L2 cache relative to the peak utilization.
tex_utilization	The utilization level of the texture cache relative to the peak utilization.
dram_utilization	The utilization level of the device memory relative to the peak utilization.
sysmem_utilization	The utilization level of the system memory relative to the peak utilization.
ldst_fu_utilization	The utilization level of the multiprocessor function units that execute load and store instructions.
alu_fu_utilization	The utilization level of the multiprocessor function units that execute integer and floating-point arithmetic instructions.
cf_fu_utilization	The utilization level of the multiprocessor function units that execute control-flow instructions.
tex_fu_utilization	The utilization level of the multiprocessor function units that execute texture instructions.
issue_slot_utilization	Percentage of issue slots that issued at least one instruction, averaged across all cycles.

Example 35 on page 58 contains the output (highlights in bold) of nvprof when profiled with the *quasirandomGenerator* sample application. The output shows the GPU utilization using all of the metrics in Table 3. Example 35 on page 58 shows that the *inverseCNDKernel* had a medium utilization of the device memory and the arithmetic function units. The *quasirandomGeneratorKernel* kernel used a maximum amount of the arithmetic function unit, but it kept low or no (idle) utilization of the other units.

Example 35 nvprof. quasirandomGenerator application output of utilization metrics (output adjusted to fit the page)

```
$ nvprof --metrics dram_utilization,l1_shared_utilization,l2_utilization,\
tex_utilization,sysmem_utilization,ldst_fu_utilization,alu_fu_utilization,\
cf_fu_utilization,tex_fu_utilization,issue_slot_utilization \
./4_Finance/quasirandomGenerator/quasirandomGenerator
./4_Finance/quasirandomGenerator/quasirandomGenerator Starting...

==7171== NVPROF is profiling process 7171, command:
./4_Finance/quasirandomGenerator/quasirandomGenerator
GPU Device 0: "Tesla K40m" with compute capability 3.5

Allocating GPU memory...
Allocating CPU memory...

<... output ommitted ...>

Shutting down...
==7171== Profiling application: ./4_Finance/quasirandomGenerator/quasirandomGenerator
==7171== Profiling result:
==7171== Metric result:
Invocations           Metric Name             Metric Description       Min       Max       Avg
Device "Tesla K40m (0)"
        Kernel: inverseCNDKernel(float*, unsigned int*, unsigned int)
         21 issue_slot_utilization         Issue Slot Utilization    60.98%    61.94%    61.55%
         21  l1_shared_utilization    L1/Shared Memory Utilization   Low (1)   Low (1)   Low (1)
         21         l2_utilization              L2 Cache Utilization   Low (2)   Low (2)   Low (2)
         21        tex_utilization       Texture Cache Utilization  Idle (0)  Idle (0)  Idle (0)
         21       dram_utilization       Device Memory Utilization   Mid (4)   Mid (4)   Mid (4)
         21     sysmem_utilization       System Memory Utilization  Idle (0)   Low (1)  Idle (0)
         21    ldst_fu_utilization  Load/Store Function Unit Utili   Low (1)   Low (1)   Low (1)
         21     alu_fu_utilization  Arithmetic Function Unit Utili   Mid (4)   Mid (4)   Mid (4)
         21      cf_fu_utilization  Control-Flow Function Unit Uti   Low (2)   Low (2)   Low (2)
         21     tex_fu_utilization  Texture Function Unit Utilizat  Idle (0)  Idle (0)  Idle (0)
        Kernel: quasirandomGeneratorKernel(float*, unsigned int, unsigned int)
         21 issue_slot_utilization         Issue Slot Utilization    51.74%    51.97%    51.91%
         21  l1_shared_utilization    L1/Shared Memory Utilization   Low (1)   Low (1)   Low (1)
         21         l2_utilization              L2 Cache Utilization   Low (1)   Low (1)   Low (1)
         21        tex_utilization       Texture Cache Utilization  Idle (0)  Idle (0)  Idle (0)
         21       dram_utilization       Device Memory Utilization   Low (2)   Low (2)   Low (2)
         21     sysmem_utilization       System Memory Utilization  Idle (0)   Low (1)  Idle (0)
         21    ldst_fu_utilization  Load/Store Function Unit Utili   Low (1)   Low (1)   Low (1)
         21     alu_fu_utilization  Arithmetic Function Unit Utili  Max (10)  Max (10)  Max (10)
         21      cf_fu_utilization  Control-Flow Function Unit Uti   Low (1)   Low (1)   Low (1)
         21     tex_fu_utilization  Texture Function Unit Utilizat  Idle (0)  Idle (0)  Idle (0)
```

The results in Example 35 show that it would be beneficial to lower the use of the function unit by the *quasirandomGeneratorKernel* kernel because it has been the most accessed unit. Therefore, Example 36 on page 59 shows the output of **nvprof** when profiling the application to generate metrics of a single and a double-precision floating point operation. Example 36 on page 59 shows in bold that the kernel ran only a single-precision floating point multiply operation.

Example 36 nvprof. quasirandomGenerator application output of arithmetic unit metrics (output adjusted to fit the page)

```
$ nvprof --kernels ::quasirandomGeneratorKernel:1 --metrics \
flops_sp_special,flops_dp_fma,flops_dp_mul,flops_dp_add,flops_sp_fma,flops_sp_mul,flops_sp_add \
./4_Finance/quasirandomGenerator/quasirandomGenerator
./4_Finance/quasirandomGenerator/quasirandomGenerator Starting...

==8079== NVPROF is profiling process 8079, command:
./4_Finance/quasirandomGenerator/quasirandomGenerator
GPU Device 0: "Tesla K40m" with compute capability 3.5

Allocating GPU memory...
Allocating CPU memory...

<... output ommitted ...>

Shutting down...
==8079== Profiling application: ./4_Finance/quasirandomGenerator/quasirandomGenerator
==8079== Profiling result:
==8079== Metric result:
Invocations            Metric Name              Metric Description       Min       Max       Avg
Device "Tesla K40m (0)"
        Kernel: quasirandomGeneratorKernel(float*, unsigned int, unsigned int)
          1           flops_sp_add               FLOPS(Single Add)         0         0         0
          1           flops_sp_mul               FLOPS(Single Mul)   3145728   3145728   3145728
          1           flops_sp_fma               FLOPS(Single FMA)         0         0         0
          1           flops_dp_add               FLOPS(Double Add)         0         0         0
          1           flops_dp_mul               FLOPS(Double Mul)         0         0         0
          1           flops_dp_fma               FLOPS(Double FMA)         0         0         0
          1       flops_sp_special           FLOPS(Single Special)         0         0         0
```

As mentioned before, hardware events can be used to expand performance analysis, although the **nvprof**-provided metrics are useful for identifying the area of potential improvements.

CUDA debug tools

The CUDA 5.5 toolkit for POWER8 provides a set of tools for debugging that can help locate issues in parallel applications:

- CUDA binary utilities
- CUDA Memcheck
- CUDA GDB

CUDA binary utilities

The CUDA binary utilities include the **cuobjdump** and **nvdisasm** tools, which are designed to inspect binary objects (**cuobjdump**), and disassemble them (**nvdisasm**).

Use **cuobjdump** to extract and present, in human-readable format, information about both host and cubin object files. It has many options for gathering information about only certain segments of the object file, for instance, any Executable and Linking Format (ELF) header. Example 37 on page 60 shows the standard output of **cuobjdump** for an executable file.

Example 37 cuobjdump standard output

```
$ cuobjdump helloCUDA/Debug/helloCUDA

Fatbin elf code:
================
arch = sm_20
code version = [1,7]
producer = <unknown>
host = linux
compile_size = 64bit
identifier = ./main.o

Fatbin elf code:
================
arch = sm_30
code version = [1,7]
producer = cuda
host = linux
compile_size = 64bit
has debug info
compressed
identifier = ../main.cu

Fatbin ptx code:
================
arch = sm_30
code version = [3,2]
producer = cuda
host = linux
compile_size = 64bit
has debug info
compressed
identifier = ../main.cu
ptxasOptions =  -g --dont-merge-basicblocks --return-at-end
```

Use the **`nvdisasm`** tool for:

- Disassembling cubin files
- Showing control flow

Notice that the tool does not allow the host code to disassemble the files.

CUDA memcheck

The **`cuda-memcheck`** is a suite of tools designed to identify memory-related issues in CUDA applications. You can execute the tool in different modes using the tool option, and by passing in either of the following values:

- memcheck
  - Analyzes the correctness of memory accesses. This includes:
    - Checking allocations on device heap
    - Detecting memory leaks
- racecheck
  - Checks the race conditions on accesses to shared memory

The **memcheck** output in Example 38 shows an error caused by thread (ID=255 in block 0) that tried to access 4 bytes out of the shared memory bounds. It also detected a problem with the CUDA API call to the *cudaThreadSynchronize* function.

Example 38 cuda-memcheck memory access checking

```
cuda-memcheck helloCUDA/Debug/helloCUDA
========= CUDA-MEMCHECK
========= Invalid __global__ read of size 4
=========     at 0x000000e8 in /home/wainersm/helloCUDA/Debug/../main.cu:39:bitreverse(void*)
=========     by thread (255,0,0) in block (0,0,0)
=========     Address 0x43046c0400 is out of bounds
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame:/usr/lib/powerpc64le-linux-gnu/libcuda.so (cuLaunchKernel + 0x24c) 
[0x152cfc]
=========     Host Frame:/usr/local/cuda-5.5-power8/lib64/libcudart.so.5.5 [0xa7f8]
=========     Host Frame:/usr/local/cuda-5.5-power8/lib64/libcudart.so.5.5 (cudaLaunch + 0x184) 
[0x39c04]
=========     Host Frame:helloCUDA/Debug/helloCUDA [0x13cc]
=========     Host Frame:helloCUDA/Debug/helloCUDA [0xba0]
=========     Host Frame:/lib/powerpc64le-linux-gnu/libc.so.6 [0x24e80]
=========     Host Frame:/lib/powerpc64le-linux-gnu/libc.so.6 (__libc_start_main + 0xc4) 
[0x25074]
=========
========= Program hit error 30 on CUDA API call to cudaThreadSynchronize 
Error unknown error at line 59 in file ../main.cu
=========     Saved host backtrace up to driver entry point at error
=========     Host Frame:/usr/lib/powerpc64le-linux-gnu/libcuda.so [0x36c39c]
=========     Host Frame:/usr/local/cuda-5.5-power8/lib64/libcudart.so.5.5 
(cudaThreadSynchronize + 0x19c) [0x3427c]
=========     Host Frame:helloCUDA/Debug/helloCUDA [0xba4]
=========     Host Frame:/lib/powerpc64le-linux-gnu/libc.so.6 [0x24e80]
=========     Host Frame:/lib/powerpc64le-linux-gnu/libc.so.6 (__libc_start_main + 0xc4) 
[0x25074]
=========
========= ERROR SUMMARY: 2 errors
```

CUDA GDB

The **cuda-gdb** command-line debugger, which is provided by the CUDA toolkit 5.5, is based on GNU GDB 7.2 implementation, allows debugging of both host and device portions of code.

cuda-gdb includes the same **gdb** command set, but also implements the specific CUDA commands, **info cuda**, **cuda**, and **set cuda**, which are operations that show information about activities (**info cuda**), set focus on context (**cuda**), and set debugger configuration (**set cuda**).

Example 39 on page 62 shows the use of ordinary **gdb** and CUDA-specific commands, acting in the same debug session. One breakpoint is set in *bitreverse* kernel and halt execution. Context information for devices, threads, and kernels is accessed with the **info cuda <option>** commands. The thread focus is switched with the **cuda thread** command. The **info frame** command, for example, is triggered to demonstrate conventional **gdb** commands.

Example 39 cuda-gdb: a simple walk-through session

```
$ cuda-gdb ./helloCUDA/Debug/helloCUDA
NVIDIA (R) CUDA Debugger
6.0 release
<... output ommitted ...>
Reading symbols from /home/wainersm/helloCUDA/Debug/helloCUDA...done.
(cuda-gdb) start
Temporary breakpoint 1 at 0x10000acc: file ../main.cu, line 46.
Starting program: /home/wainersm/helloCUDA/Debug/helloCUDA
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/powerpc64le-linux-gnu/libthread_db.so.1".

Temporary breakpoint 1, main () at ../main.cu:46
46    void *d = NULL;
(cuda-gdb) info stack
#0  main () at ../main.cu:46
(cuda-gdb) next
45 int main(void) {
(cuda-gdb) next
46    void *d = NULL;
(cuda-gdb) next
51       idata[i] = (unsigned int) i;
(cuda-gdb) next
53    CUDA_CHECK_RETURN(cudaMalloc((void**) &d, sizeof(int) * WORK_SIZE));
(cuda-gdb) next
[New Thread 0x3fffb6b7f180 (LWP 6018)]
54    CUDA_CHECK_RETURN(
(cuda-gdb) next
57    bitreverse<<<1, WORK_SIZE, WORK_SIZE * sizeof(int)>>>(d);
(cuda-gdb) info frame
Stack level 0, frame at 0x3fffffff100:
 pc = 0x10000b8c in main (../main.cu:57); saved pc 0x3fffb7d74e80
 source language c++.
 Arglist at 0x3ffffffe820, args:
 Locals at 0x3ffffffe820, Previous frame's sp is 0x3fffffff100
 Saved registers:
  r22 at 0x3fffffff0b0, r23 at 0x3fffffff0b8, r24 at 0x3fffffff0c0, r25 at
0x3fffffff0c8, r26 at 0x3fffffff0d0,
  r27 at 0x3fffffff0d8, r28 at 0x3fffffff0e0, r29 at 0x3fffffff0e8, r30 at
0x3fffffff0f0, r31 at 0x3fffffff0f8,
  pc at 0x3fffffff110, lr at 0x3fffffff110
(cuda-gdb) info cuda devices
  Dev Description SM Type SMs Warps/SM Lanes/Warp Max Regs/Lane Active SMs Mask
    0      GK110B   sm_35  15       64         32           256 0x00000000
    1      GK110B   sm_35  15       64         32           256 0x00000000
(cuda-gdb) info cuda kernels
No CUDA kernels.
(cuda-gdb) info breakpoints
No breakpoints or watchpoints.
(cuda-gdb) break main.cu:38
Breakpoint 5 at 0x103f42a0: file ../main.cu, line 38.
(cuda-gdb) list 38
33
34 /**
35  * CUDA kernel function that reverses the order of bits in each element of the
array.
36  */
37 __global__ void bitreverse(void *data) {
38   unsigned int *idata = (unsigned int*) data;
39   idata[0] = bitreverse(idata[threadIdx.x]);
40 }
41
42 /**
(cuda-gdb) info breakpoints
Num     Type           Disp Enb Address            What
5       breakpoint     keep y   0x00000000103f42a0 in bitreverse(void*) at
../main.cu:38
(cuda-gdb) continue
Continuing.
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (0,0,0), device
0, sm 14, warp 0, lane 0]

Breakpoint 5, bitreverse<<<(1,1,1),(256,1,1)>>> (data=0x43046c0000) at
../main.cu:39
39   idata[0] = bitreverse(idata[threadIdx.x]);
(cuda-gdb) next
40 }
(cuda-gdb) info cuda threads
  BlockIdx ThreadIdx To BlockIdx ThreadIdx Count         Virtual PC   Filename
Line
Kernel 0
*  (0,0,0)   (0,0,0)     (0,0,0)  (31,0,0)    32 0x00000000103f43a0 ../main.cu
40
   (0,0,0)  (32,0,0)     (0,0,0) (255,0,0)   224 0x00000000103f42a0 ../main.cu
39
(cuda-gdb) cuda thread 10
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (10,0,0), device
0, sm 14, warp 0, lane 10]
40 }
(cuda-gdb) cuda thread
thread (10,0,0)
(cuda-gdb) info cuda devices
  Dev Description SM Type SMs Warps/SM Lanes/Warp Max Regs/Lane Active SMs Mask
*   0      GK110B   sm_35  15       64         32           256 0x00004000
    1      GK110B   sm_35  15       64         32           256 0x00000000
(cuda-gdb) cuda sm
sm 14
(cuda-gdb) continue
Continuing.
Input value: 0, device output: 254, host output: 0
Input value: 1, device output: 1, host output: 128
Input value: 2, device output: 2, host output: 64
Input value: 3, device output: 3, host output: 192
<... output omitted ...>
[Thread 0x3fffb6b7f180 (LWP 6018) exited]
[Inferior 1 (process 5985) exited normally]
(cuda-gdb)
The program is not being run.
```

Remote debugging is possible using **`cuda-gdbserver`** (which is also delivered with the CUDA toolkit 5.5). **cuda-gdbserver** behaves just like **gdb-server**; it allows sessions to be handled over an SSH or FTP connection.

Another alternative to debugging CUDA applications is to use the Nsight debugger, which is integrated with the **`cuda-gdb`** graphically. The Nsight debugger is covered in the section titled "Nsight" on page 44.

The reference documentation of the **`cuda-gdb`** debugger is available at the following site:

`http://docs.nvidia.com/cuda/cuda-gdb`

Application examples

Many existing application programs can leverage the computational power of NVIDIA GPUs. Systems equipped with GPUs are very effective for accelerating compute intensive applications. In this section, GROMACS and LAMMPS provide examples for demonstrating how to install, configure, and run CUDA applications on a GPU accelerated POWER8 based server.

GROMACS

GROningen MAchine for Chemical Simulations (GROMACS) is a molecular dynamics package that is primarily designed for simulations of proteins, lipids, and nucleic acids. For more information about GROMACS, refer to the following website:

`http://www.gromacs.org`

Installation of GROMACS

Before installation, download and unpack the GROMACS source code from the GROMACS website. In this section, GROMACS 4.6.7 provides an example. Ensure that the GCC compiler is up to date, and that the latest level of CMake is installed.

Request to the system administrator to install *CMake*, *libfftw3-3*, and *libfftw3-dev* by using the Ubuntu **`apt-get`** command:

```
$ sudo apt-get install cmake libfftw3-3 libfftw3-dev
```

Example 40 shows the commands needed to install GROMACS version 4.6.7.

Example 40 The commands to install GROMACS

```
$ mkdir -p $HOME/install/gromacs467
$ tar xfz gromacs-4.6.7.tar.gz
$ cd gromacs-4.6.7
$ mkdir build
$ cd build
$ PREFIX=$HOME/local/gromacs/4.6.7
$ cmake .. -DGMX_GPU=ON -DGMX_PREFER_STATIC_LIBS=ON -DCMAKE_INSTALL_PREFIX=$PREFIX
$ make
$ make install
$ source $PREFIX/bin/GMXRC
```

Running GROMACS simulations

Some test data sets are available at the GROMACS website. In this section, the *water_GMX50_bare.tar.gz* data set provides an example. You can download the data set from the following site:

ftp://ftp.gromacs.org/pub/benchmarks/water_GMX50_bare.tar.gz

In the water_GMX50_bar.tar.gz data set, there are three types of input files:

- The *.mdp* file defines the parameters for running the simulation.
- The *.top* file extension stands for topology.
- the *.gro* file is a fixed-column coordinate file format, first used in the GROMOS simulation package.

To run the simulation, you need to generate a `.tpr` file. The `.tpr` file extension stands for portable binary run input file. The `.tpr` file contains the starting structure of your simulation (coordinates and velocities), the molecular topology, and all the simulation parameters. The `.tpr` file is generated by the **`grompp`** command and then executed by the **`mdrun`** command to perform the simulation. Example 41 shows how to prepare the data set for the simulation.

Example 41 Commands to prepare a GROMACS simulation data set

```
$ tar zxf water_GMX50_bare.tar.gz
$ cd water-cut1.0_GMX50_bare/1536
$ grompp -f pme.mdp
```

The final step is to run the simulation. Run the following command in the data set directory:

```
$ mdrun -s topol.tpr -npme 0 -resethway -noconfout -nb gpu -nsteps 1000 -v
```

The output of the command contains the GPU information for the run, as shown in Example 42.

Example 42 Output of a GROMACS simulation run

```
<... output omitted ...>
Reading file topol.tpr, VERSION 4.6.7 (single precision)
Changing nstlist from 10 to 40, rlist from 1 to 1.101

Overriding nsteps with value passed on the command line: 1000 steps, 2.000 ps

Using 2 MPI threads
Using 80 OpenMP threads per tMPI thread

2 GPUs detected:
  #0: NVIDIA Tesla K40m, compute cap.: 3.5, ECC: yes, stat: compatible
  #1: NVIDIA Tesla K40m, compute cap.: 3.5, ECC: yes, stat: compatible

2 GPUs auto-selected for this run.
Mapping of GPUs to the 2 PP ranks in this node: #0, #1

starting mdrun 'Water'
1000 steps,      2.0 ps.
step   80: timed with pme grid 200 200 200, coulomb cutoff 1.000: 1591.3 M-cycles
step  160: timed with pme grid 168 168 168, coulomb cutoff 1.187: 1333.3 M-cycles
step  240: timed with pme grid 144 144 144, coulomb cutoff 1.384: 1655.6 M-cycles
<... output omitted ...>
```

Support for the message passing interface (MPI) was not enabled at the time of compilation. However, GROMACS has its internal thread-based implementation of MPI. For this reason, **mdrun** reports the usage of MPI when running on a two-socket IBM Power Systems S824L testbed. For more details about the available options for acceleration and parallelization of GROMACS, refer to the following web page:

http://www.gromacs.org/Documentation/Acceleration_and_parallelization

LAMMPS

Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) is a classical molecular dynamics simulation code designed to run efficiently on parallel computers. LAMMPS runs on single processors or in parallel using message-passing techniques and a spatial-decomposition of the simulation domain. For more information about LAMMPS, refer to the following website:

http://lammps.sandia.gov

Installation of LAMMPS

Before installation, download and unpack the LAMMPS source code from the LAMMPS website:

http://lammps.sandia.gov/download.html#tar

In this publication, the LAMMPS stable version (30 Oct 2014) is downloaded and installed.

Ask the system administrator to install *CMake*, *libfftw3-3*, and *libfftw3-dev* by using the Ubuntu **apt-get** command:

```
$ sudo apt-get install cmake libfftw3-3 libfftw3-dev
```

Example 43 shows the detailed commands that are used to deploy LAMMPS.

Example 43 Deploying LAMMPS

```
$ tar xzf lammps.tar.gz
$ cd ./lammps-30Oct14
$ export CUDA_HOME=/usr/local/cuda
$ export PATH=$PATH:/usr/local/cuda/bin
$ cd src/
$ cp MAKE/Makefile.serial MAKE/MINE/Makefile.auto
$ python Make.py -p cuda -cuda arch=20 -o cuda lib-cuda exe
Installing packages ...
Action lib-cuda ...
building cuda library ...
Created lib/cuda library
Action exe ...
Created src/MAKE/MINE/Makefile.auto
building serial STUBS library ...
Created src/STUBS/libmpi_stubs.a
Created lmp_cuda in /home/weili/lammps-30Oct14/src
$
```

The binary **lmp_cuda** is ready for running LAMMPS simulations.

Running LAMMPS simulations

Data sets are available on the LAMMPS website at:

http://lammps.sandia.gov/bench.html

In this section, the *in.lj* data set is used for LAMMPS simulation. Download the data set from the following website:

http://lammps.sandia.gov/inputs/in.lj.txt

Download the file *in.lj.txt* and rename it to *in.lj*, then run the following commands for the simulation:

```
$ <LAMMPS_installation_dir>/src/lmp_cuda -sf cuda -c on -v g 2 -v x 64 -v y 64 -v
z 64 -v t 1000 < in.lj
```

The output of the commands contains the GPU information for the run, as shown in Example 44.

Example 44 Output of a LAMMPS simulation

```
# Using LAMMPS_CUDA
USER-CUDA mode is enabled (../lammps.cpp:475)
Lattice spacing in x,y,z = 1.6796 1.6796 1.6796
Created orthogonal box = (0 0 0) to (33.5919 33.5919 33.5919)
# CUDA: Activate GPU
# Using device 0: Tesla K40m
  1 by 1 by 1 MPI processor grid
Created 32000 atoms
# CUDA: Using precision: Global: 8 X: 8 V: 8 F: 8 PPPM: 8
Setting up run ...
# CUDA: VerletCuda::setup: Upload data...
Test TpA
Test BpA

# CUDA: Timing of parallelisation layout with 10 loops:
# CUDA: BpA TpA
 0.052649 0.011521
# CUDA: Total Device Memory usage post setup: 162.921875 MB
<... output ommitted ...>
```

Authors

This paper was produced by a team of specialists from around the world, working at the IBM International Technical Support Organization (ITSO), Poughkeepsie Center.

Dino Quintero is a Complex Solutions Project Leader and an IBM Senior Certified IT Specialist with the ITSO in Poughkeepsie, NY. His areas of knowledge include enterprise continuous availability, enterprise systems management, system virtualization, technical computing, and clustering solutions. He is an Open Group Distinguished IT Specialist. Dino holds a Master of Computing Information Systems degree and a Bachelor of Science degree in Computer Science from Marist College.

Wei Li is a Staff Software Engineer in Blue Gene and a member of the Fortran compiler test team with the IBM Canada Software Laboratory, Toronto. He joined IBM in 2008 and has nine years of experience in the IT industry. His areas of expertise include AIX, Linux, Blue Gene, IBM System p® hardware, and High Performance Computing software solutions. He holds an IBM Certification in pSeries AIX 6 System Support. He holds a Bachelor of Engineering and Master of Engineering degree in Information and Communication Engineering from the University of Science and Technology of China.

Wainer dos Santos Moschetta is a Staff Software Engineer in the IBM Linux Technology Center, Brazil. He initiated and formerly led the IBM Software Development Kit (SDK) project for the IBM PowerLinux™ project. He has six years of experience with designing and implementing software development tools for Linux on IBM platforms. Wainer holds a Bachelor degree in Computer Science from the University of São Paulo. He co-authored IBM Redbooks® publications SG24-8075 and SG24-8171, has published articles and videos for the IBM DeveloperWorks website, and contributes to the IBM PowerLinux technical community blog.

Mauricio Faria de Oliveira is a Staff Software Engineer in the Linux Technology Center at IBM Brazil. His areas of expertise include Linux performance and Debian/Ubuntu distributions on IBM Power Systems, having worked with official benchmark publications and early development of Debian on little-endian 64-bit PowerPC. Mauricio holds a Master of Computer Science and Technology degree and a Bachelor of Engineering degree in Computer Engineering from Federal University of Itajubá.

Alexander Pozdneev is a Technical Consultant in STG Lab Services at IBM East Europe/Asia, Moscow, Russia. He has 11 years of experience in HPC. He holds a Ph.D. degree in Mathematical Modeling, Numerical Methods, and Software from Lomonosov Moscow State University. His areas of expertise include parallel computing and application performance optimization.

Thanks to the following people for their contributions to this project:

Ella Bushlovic
IBM International Technical Support Organization, Poughkeepsie Center

Ruzhu Chen, Gerald Gantz, Holly Gross, Brian Hart, Xinghong He, John Lemek, Bill LePera, Joan McComb, Mark Perez, David Shapiro, Tzy-hwa K Tzeng, Duane Witherspoon
IBM US

Guang Cheng Li, Xiao Lu Qi, Si Si Shen, Yu Song, Er Tao Zhao
IBM China

Now you can become a published author, too!

Stay connected to IBM Redbooks

Notices

This document REDP-5169-00 was created or updated on March 26, 2015.

Send us your comments in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:
  `ibm.com/redbooks`
- Send your comments in an email to:
  `redbooks@us.ibm.com`
- Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400 U.S.A.

Redpaper™

Trademarks