Howto Build Your Desktop Supercomputer

Administrative Recipes to Build Your

HPMC

(High Performance Manycore Computer)

SAMPLE BOOK PREVIEW

Bassem W. Jamaleddine

TCND Inc.
POBox 680
New York, NY 10156
ph. (212) 203 4379
fax (212) 208 3039
www.tcnd.com

Books for SetSphere Shops

SetSphere - ASPL User Manual
SetSphere - ASPL Operations Guide
SetSphere - ASPL USer Guide

This book was written, edited, designed, and produced
by Bassem Jamaleddine.

SAMPLE BOOK PREVIEW

© 2026, Total Computing & Network Design, Inc.
All rights reserved.

ISBN 979-8-16-112122-1

Contents

Preface iii
Chapter 1 HPMC Calculator 3
1-1 HPMC Calculator 4
Chapter 2 Xeon Phi MIC 21
2-1 Configuring the Xeon Phi MIC Coprocessor 22
2-2 The MIC Bootstrap Process 31
2-3 Sharing Files Between the Host and the MIC 36
2-4 Performance Programming for MIC 40
2-5 DDD - Data Display Debugger for the MIC 45
2-6 Programming and Running by MIC Native Loading 47
2-7 Matrix Multiplication Offloading to MIC 54
2-8 Measuring Gigaflops for Matrix Multiplication on the MIC 65
Chapter 3 Computer Organisation of the MIC 71
3-1 MIC Cores Threads and Cache Organization 72
3-2 MIC Superscalar Vector Processor a Programmer View 75
3-3 MIC Vector Operations a Programmer View 87
3-4 MIC Architecture Superscalar and Pipelines 97
Chapter 4 CUDA Programming on your HPMC GPU 107
4-1 System Configuration for the NVIDIA GPU on Your HPMC 108
4-2 CUDA Programming for your HPMC GPU 118
4-3 Measuring Practical Gigaflops for Matrix Multiplication on the GPU 120
4-4 CUDA Fortran 127
4-5 CUDA Application for Your HPMC GPU 131
4-6 GPU Programming with CAFFE 135
4-7 GPU Programming with Theano 138
4-8 QUDA: Accelerated lattice Quantum Chromodynamics Calculations on the GPU 142
Chapter 5 Configuring Infiniband for RDMA 153
5-1 Configuring Infiniband for RDMA 154
Chapter 6 RDMA Over Infiniband Connectivity 163
6-1 RDMA Over Infiniband Connectivity 164
6-2 MPI Programming with Fabric MPIRUN Offloading to MIC 170
Appendices
Appendix A Logging in to the MIC 180
Appendix B Troubleshooting the MIC 182
Appendix C Troubleshootig the CUDA Configuration 183
Appendix D Shared Libraries of an Offloaded Program 184
Appendix E Simple OMP Program 186
Appendix F MPI Threads and Ranks and Context Switching 188
Appendix G dint command to display system interrupts 193
Appendix H dcpus command to display the system stats 196
Appendix I gpdata3 200
Appendix J How to display the system interrupts of your HPMC 201
Appendix K How to display the interrupts of the MIC 203
Appendix L FLOPS 204
Appendix M Installing Torch cuDNN and PyTorch 207

SAMPLE BOOK PREVIEW

Appendix N Historical Perspective on High Performance Switch 211
Appendix O Historical Perspective on Multiply Add Fused 216
Appendix P Xeon Phi and MIC versus GPU 219
Glossary 223
Index 228

SAMPLE BOOK PREVIEW
SAMPLE BOOK PREVIEW

HPMC USER GUIDE
HPMC 1.00

PREFACE

9 November 2026

Congratulations on your selection of the HPMC calculator:
HPMC = High Peformance Manycore Computer, it is powered by the Linux operating system. This HPMC-calculator is the first in digital computation technology that brings high performance manycore calculation to your desktop.

The HPMC has a motherboard that we will refer to it as the mainboard. The HPMC is the host to a MIC and to a GPU; both the MIC and the GPU are inserted in the mainboard of the HPMC.

In this user guide we refer to the UNIX shell prompt with a hash as shown here

```
#  systemctl status mpss
```

Since a host may be serving a MIC coprocessor that is plugged in its mainboard and running a micro Linux OS, for clarity, and to distinguish the shell prompt of the host, say HPMC7, from the shell of its MIC mic0, we will use the following notation.
The shell prompt of host named HPMC7 is shown as:

SAMPLE BOOK PREVIEW

```
23:25 root@HPMC7: ~ #  ssh mic0
```

The shell prompt of the MIC mic0 being served by HPMC7 is shown as:

SAMPLE BOOK PREVIEW

```
(uos/HPMC7) 23:27 root@mic0: ~ #  cat /etc/hosts
```

That is the hash represents the shell prompt of the host name mic0 that is being served by HPMC7. Notice the parentheses marking the name of the host where mic0 is being served. This command is being executed on the MIC itself.
If the order of command execution is important, then the commands on the MIC mic0 are numbered as follows:

```
1) - (uos/HPMC7) 23:29 root@mic0: ~ #  cat /etc/hosts
```

```
2) - (uos/HPMC7) 23:31 root@mic0: ~ #  cat /root/.profile
```

The formatting of the command was made possible with a specialized formatter PPAGES that I wrote to convey information in a distributed UNIX environment that depends on the shell command prompt. For the MIC installed on HPMC, the root account should have the following command prompt spec defined in its /root/.profile:

```
PS1='\[\033[01;37;46m\](HPMC) \A \u@\h: \w # \[\033[00m\] '
```

For the MIC installed on HPMC7, the root account should have the following command prompt spec defined in its /root/.profile:

```
PS1='\[\033[01;37;46m\](HPMC7) \A \u@\h: \w # \[\033[00m\] '
```

The commands shown in this manual have been captured on three HPMCs: hpmc, hpmc7, and hpmc9. These are also the hostnames of the HPMCs. Each HPMC is hosting an Nvidia GPU and a MIC mic0. The three MICs are identical and uses Knights Corner architecture. However, the HPMCs have different mainboards, eg. different processor and memory are on the mainboard. In particular, hpmc7 mainboard has a Knights Landing Xeon Phi processor with 68 cores and 272 threads.

Although this manual is made public in November 2026, it forms the pages of many system integrations that I worked on between 2015 and 2022 to build a desktop that can serve as a High Performance Manycore Computer (HPMC). A desktop that is similar to the Personal Computer (PC) of the 80s: a computer that you can put on the desk. Eventually this desktop ended up as a supercomputer that is cased in a chassis whose dimensions are 14 x 14 x 8 inches.

This user guide is a compilation of pages produced in four periods:

- 2015 (6 Months): CUDA CUBE build focused on foundational CUDA GPU programming.
- 2017 (6 Months): CUDA Desktop build focused on CUDA GPU deep learning programming.
- 2019 (6 Months): HPMC build focused on adding Intel Xeon Phi MIC (Many Integrated Cores) support.
- 2022 (6 Months): HPMC-KLIB build focused on Infiniband fabric interconnectivity over RDMA.

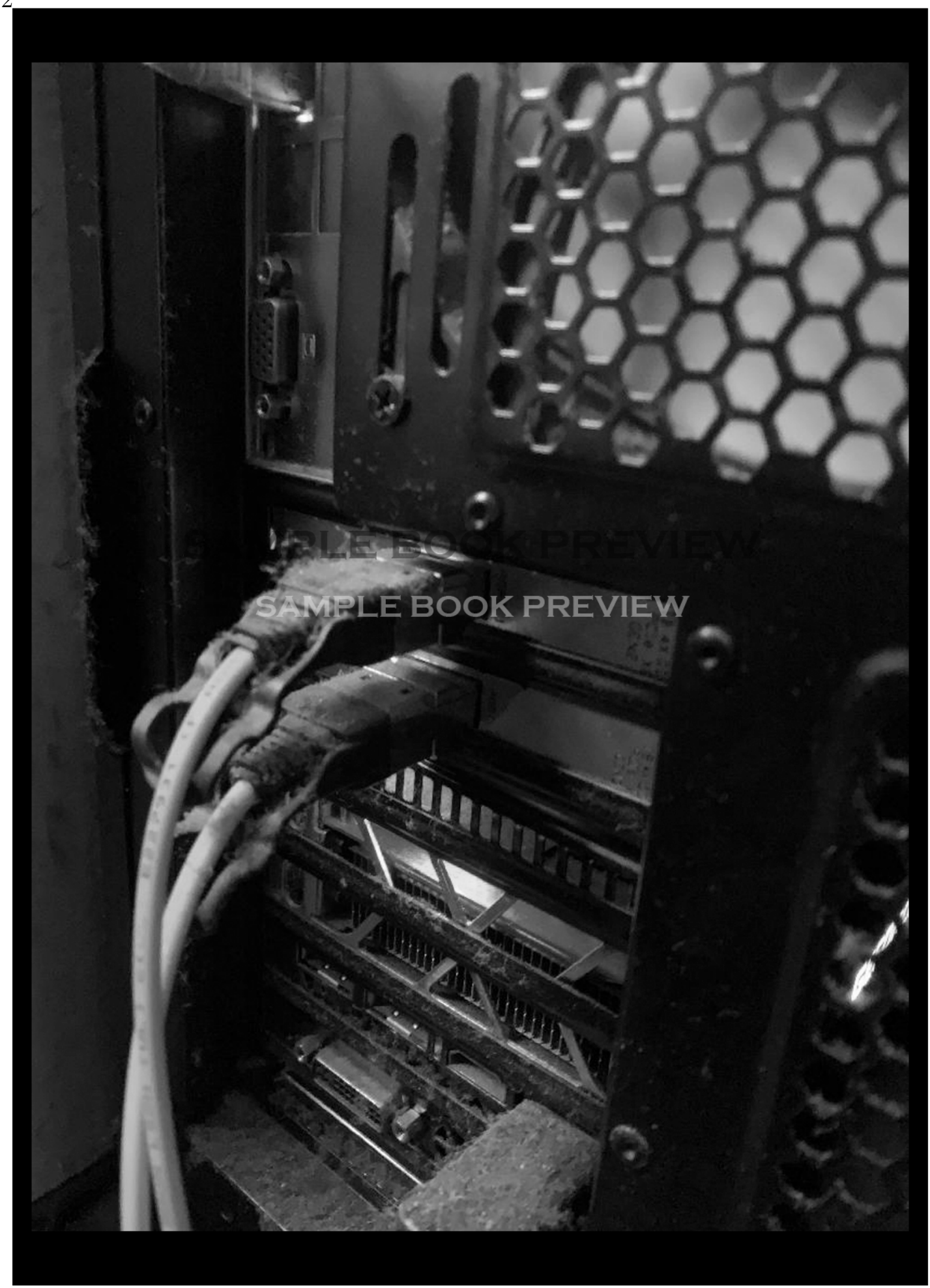
In 1995, I got the opportunity to configure the HPS (High Performance Switch based on the Vulcan chip) at IBM Watson Research Scalable POWERparallel lab where I interconnected SP2 wide nodes to be monitored with Tivoli spmon. Twenty three years later, in 2019, with discounted QLogic hardware I brought the InfiniBand switching to the home desktop computer.

Special thanks are due to Steph van Schalkwyk for providing the initial MPSS stack distribution, and to Betty Holes for providing the IBM RS/6000 PowerPC 601 system. This historic platform pioneered the Multiply-Add Fused (MAF) instruction, establishing the architectural foundation for modern high-performance computing.

Bassem W. Jamaleddine
New York, NY
November 2026

SAMPLE BOOK PREVIEW
SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW


SAMPLE BOOK PREVIEW

Chapter 1
HPMC Calculator

The HPMC Calculator is a small desktop appliance that is capable of running many threads and many cores for high performance computing. The computer is easy to program without the help of any system administrator. The HPMC is the smallest super computer that can be put on the top of your desk. Its computation power can be compared to the large computers that are farmed in large data center. Unlike the super computers that require tense system administration, the HPMC is treated like a calculator. You will power it on and start exploring with high performance computing - similar to what we used to do on the 80's desktop. Inside your HPMC are two cards: an NVIDIA GPU, and an Intel Xeon Phi. A third card, the InfiniBand HCA adapter may be available on a more advanced HPMC calculator. Hence various technology are embedded inside the HPMC, and they are all being driven to provide you with: CUDA Pogramming, Many Integrated Cores, and a DMA connectivity across multiple HPMCs. The aim of this manual is to introduce you to super computing on your HPMC. It also serves as a technical documentation to show the configuration layers of the devices in your HPMC.

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

1-1 HPMC Calculator

In this book, we will be configuring and programming an HPMC, High Performance Manycore Computer, that includes the following:

- Xeon Phi Co-Processor called a MIC coprocessor. MIC stands for Many Integrated Cores.
- GPU that is configured for CUDA programming. The GPU selected is an Nvidia GTX TITAN X.
- A Host Channel Adapater (HCA) for InfiniBand fabric connectivity to a switch, for RDMA programming. RDMA stands for Remote Direct Memory Access.

Figure 1.1.1 shows the HPMC-KLIB that includes all the three devices mentioned above. It is one of the smallest most powerful desktop x86 computer running Linux Centos; its **dimensions 14 x 14 x 8 inches**. The label KLIB stands for Knights Landing with Infiniband. This HPMC has a mainboard SuperMicro K1SPE with liquid cooling for the CPU, GPU, and high static pressure fan for the Xeon Phi MIC, and the Infiniband interface card. The K1SPE can be configured with a max memory of 192GB, in six slots each taking a Samsung PC-17000 2133Mhz 32GB Registered ECC Rank 2.

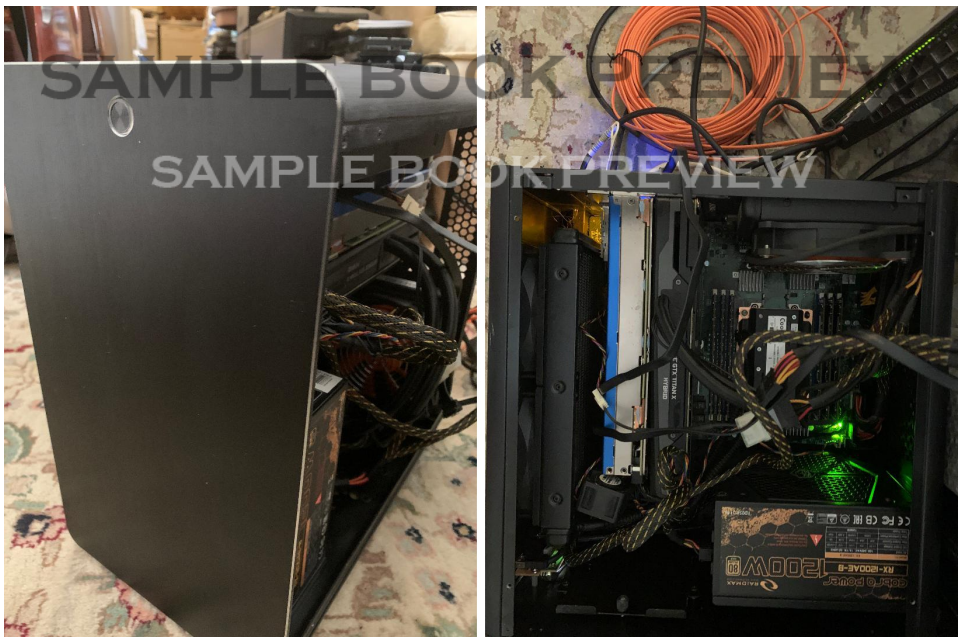

SAMPLE BOOK PREVIEW

Fig. 1.1.1 [HPMC-KLIB the smallest desktop most powerful High Performace Manycore Computer]

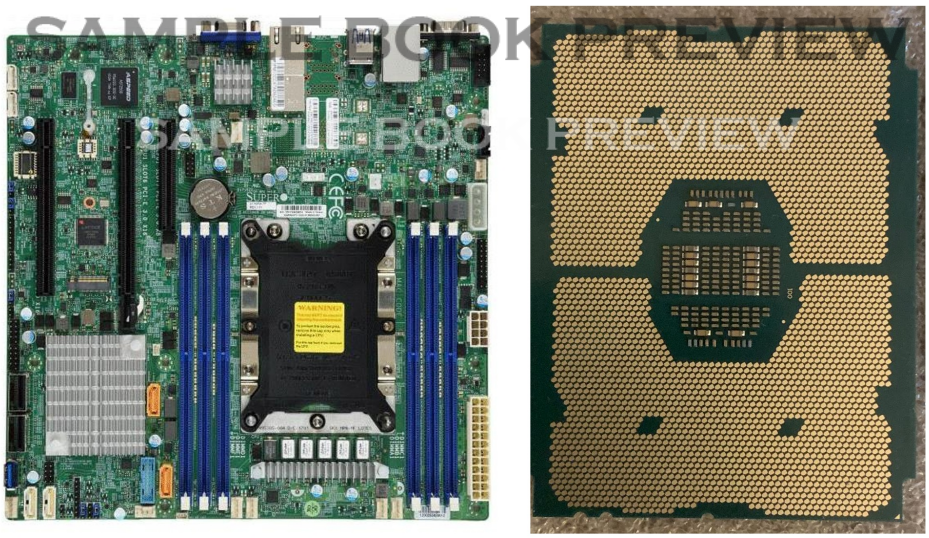
HPMC 2026 by Bassem Jamaleddine

The HPMC-KLIB uses the HYDRA case shown in Figure 1.1.2. It uses the board K1SPE and the Xeon Phi Knights Landing 7250 processor shown in Figure 1.1.3.



Fig. 1.1.2 [HYDRA Case with inside layout to accomodate water cooling]

HPMC 2026 by Bassem Jamaleddine



SAMPLE BOOK PREVIEW

Fig. 1.1.3 [K1SP Board and Xeon Phi Knights Landing Processor]

HPMC 2026 by Bassem Jamaleddine

Since the Knights Landing 7250 has 68 cores, 272 threads, running at 1.4 GHz, with a TDP 215W, we will be cooling it with CoolIt liquid cooler as shown in Figure 1.1.4. In this book the HPMC host named hpmc7 is such a computer.



SAMPLE BOOK PREVIEW

Fig. 1.1.4 [Xeon Phi 7250 on the Main Board]

HPMC 2026 by Bassem Jamaleddine

Since booting the HPMC takes few minutes, and building its kernel OS requires multiple passes and reboots, it is more reasonable to build the Linux Centos OS on a smaller HPMC (that is faster to boot). Figure 1.1.5 shows a HPMC with the board ASUS ROG MAXIMUS VIII GENE that takes less than a minute to boot. This HPMC has a hot swappable SATA drive bay. This makes it easy to build the kernel incrementally: after major changes in the kernel configuration, remove the drive from the bay, clone it and put the clone on the side. This allows you to revert quickly to the previous build when a configuration change is deemed unacceptable to boot the HPMC.

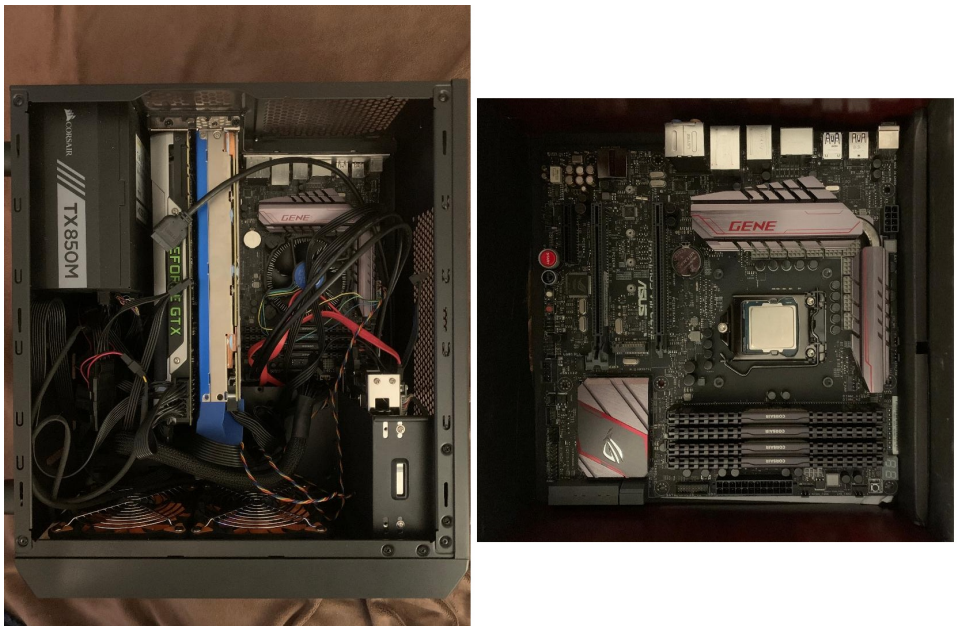


Fig. 1.1.5 [HPMC with simple board MAXIMUS VIII GENE]

HPMC 2026 by Bassem Jamaleddine

SAMPLE BOOK PREVIEW

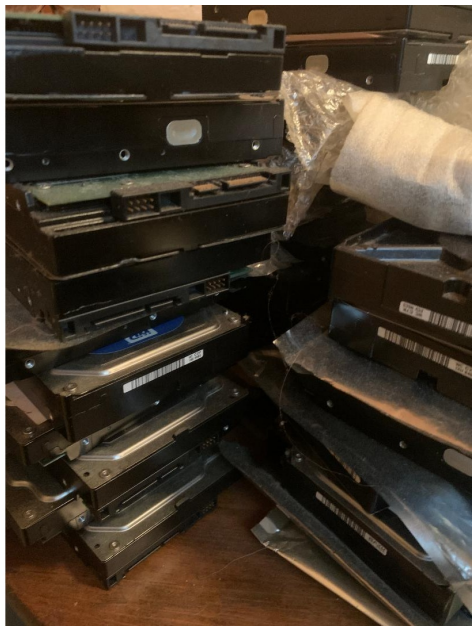
Once the OS image is built on the hard drive, we clone it and move it to a more capable HPMC. Often you do not need to recompile the OS when moving the hard drive to a new board, yet recompiling the drivers must be done on the board serving the devices, since these are compiled as loadable modules, and can be selected in the config file prior to calling the make file.

In general, an incremental build of the kernel follows this cycle:

- configure the kernel
- make the kernel
- reboot and test
- if test passes, then clone the drive, repeat this cycle if more configuration is needed.
- if test fails, then image the drive from the last clone and repeat this cycle.

Do not use virtual machines to build the kernel. Do not use copy-on-write (CoW) clone strategy to avoid cloning the physical drive. A classical build of the kernel is required for the HPMC, in particular when it comes to compiling the InfiniBand modules and their verb bindings. In addition, some of the MPSS stack needs to be recompiled. Figure 1.1.6 shows the stack of the SATA drives resulting from compiling the kernel for the final HPMC used in this book.





Building the kernel OS incremetally will result in a stack of SATA drives.
The SATA bay on the HPMC facilitates the incremental build process: “build and clone” allow the kernel builder to back out on failure.

Fig. 1.1.6 [Stack of hard drives building the OS kernel incrementally]

HPMC 2026 by Bassem Jamaleddine

SAMPLE BOOK PREVIEW

Not any board with PCIe can be used for the MIC. For the MIC to be bootable on your mainboard, the least requirement is to have a mainbaord where the BIOS allows you to protect the memory mapped MMIO above 4GB. When selecting a board for your HPMC, make sure that the BIOS has an entry to reserve the memory mapped MMIO above 4GB. The least capable board that fall in this category is the ASUS ROG MAXIMUS VIII GENE shown in Figure 1.1.5. This compact board is capable to house an Intel MIC and an Nvidia GPU.

Other boards can be used for your HPMC, for example Figure 1.1.7 shows the board ASRock X299M EXTREME 4, with max 18 cores using Intel Core i9-7980XE, that can be used for your HPMC. This board can fit in the CUBE chassis. Figure 1.1.7 shows three stacked HPMC Cubes.





Fig. 1.1.7 [HPMC Cubes using board ASRock X299M EXTREME 4]

HPMC 2026 by Bassem Jamaleddine

If the frequency of the cores matter, then you can use the Intel Xeon Platinum 8176 processor. This processor has 28 cores, 56 threads, can hit a turbo frequency 3.80GHz with an L3 38.5 MB and TDP power 165W. Figure 1.1.8 shows a Xeon Platinum 8176 processor.



SAMPLE BOOK PREVIEW

Fig. 1.1.8 [Xeon Phi Platinum 8176 28 Cores 56 Threads]

HPMC 2026 by Bassem Jamaleddine

The board SUPERMICRO X11SPH-NCTF Socket P LGA-3647 can be used for the Xeon Platinum 8176 processor. The HYDRA case can accomodate this board along the air cooler Noctua NH-D9 DX-3647 4U. Such an HPMC is shown in Figure 1.1.9.

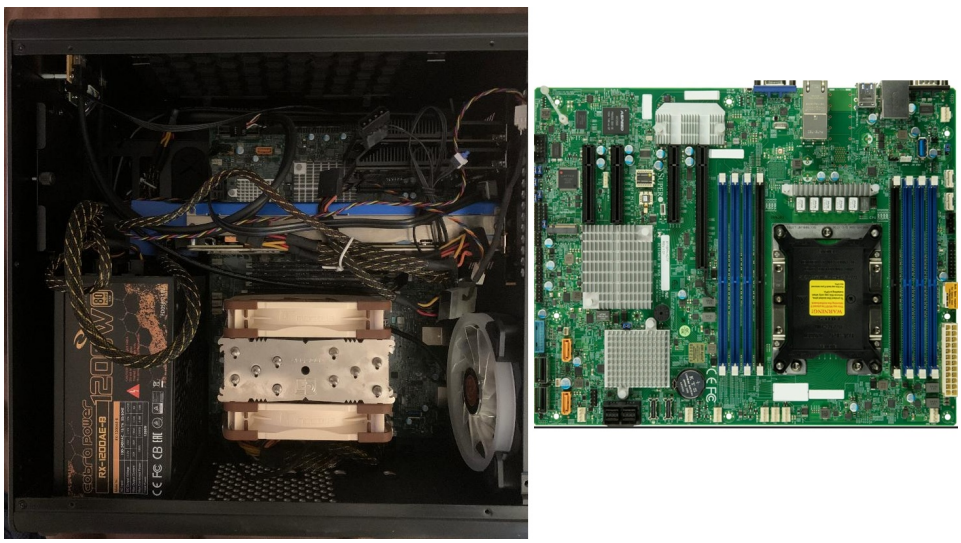
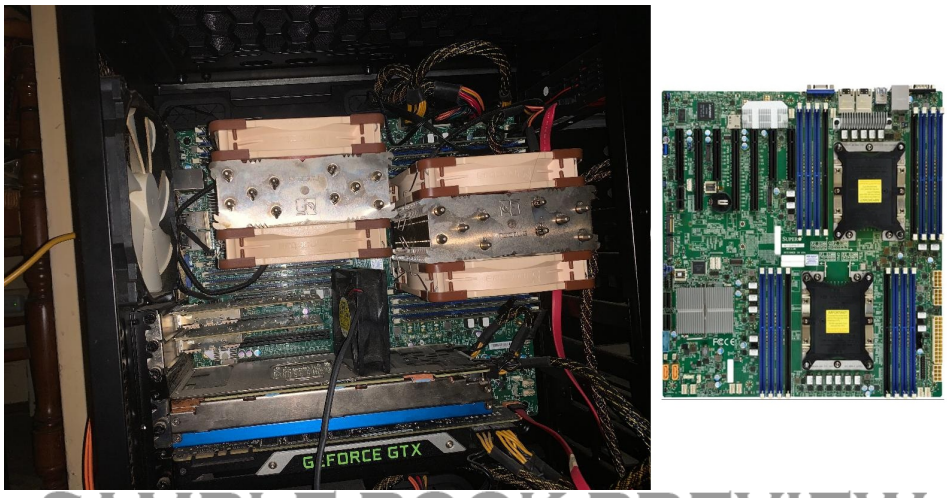


Fig. 1.1.9 [HPMC Hydra with X11SPH-NCTF]

HPMC 2026 by Bassem Jamaleddine

The board SUPERMICRO MBD-X11DPH-T-O X11 C624 DP LGA3647 XEON has dual sockets for two Xeon Platinum 8176 processors, for a total of 56 cores, and 112 threads. However, you need a tower to fit this larger board. Figure 1.1.10 shows such HPMC. This large board requires several minutes to boot. In this book, we will run InfiniBand commands and MPI programs on the host name hpmc9 that uses the aforementioned board.



SAMPLE BOOK PREVIEW

Fig. 1.1.10 [HPMC Tower Dual LGA3647 SUPERMICRO MBD-X11DPH-T-O]

HPMC 2026 by Bassem Jamaleddine

SAMPLE BOOK PREVIEW

■ The MIC

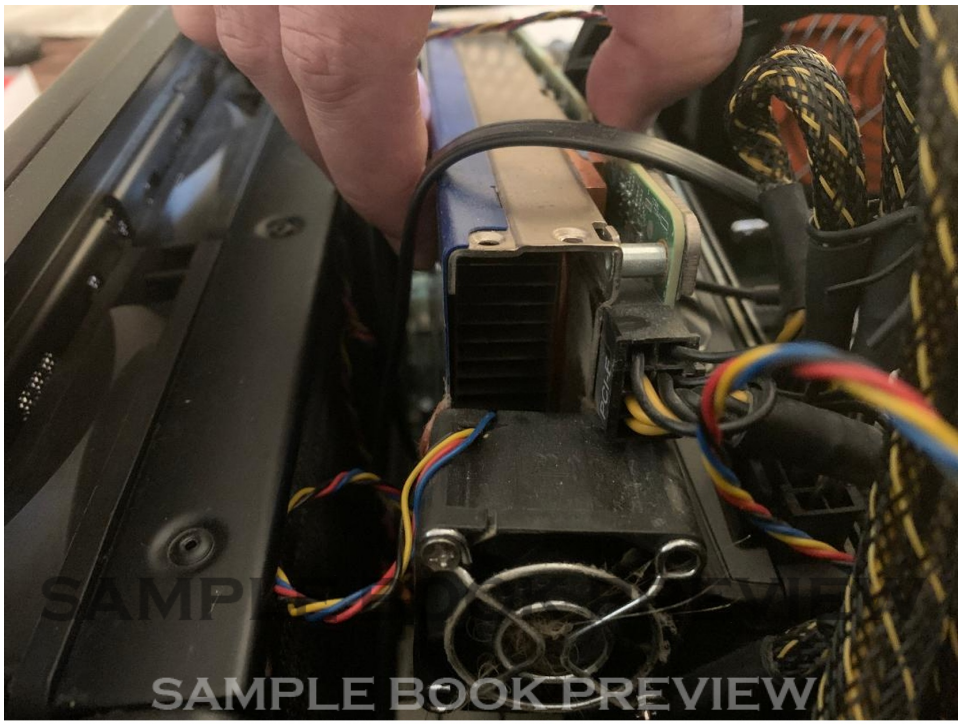
The MIC, shown in Figure 1.1.11, has 61 cores (244 threads) running at a frequency 1.1GHz requires adequate cooling. You can use high static pressure fan 40mm to push through the MIC dense tight heatsink. Figure 1.1.12 shows the MIC cooled with a 40mmx40mm fan. You can use a heavy duty heat resistant double sided tape to attach the fan to the MIC.



Fig. 1.1.11 [Intel Xeon Phi Coprocessors]

HPMC 2026 by Bassem Jamaleddine

Blower fan mounts perpendicular to the card's rear inlet, compressing the air and forcing it down the MIC heatsink. A standard shroud acts as a duct, preventing air from leaking out the sides and forcing 100% of the fan's output directly through the internal heatsink fins.


SAMPLE BOOK PREVIEW

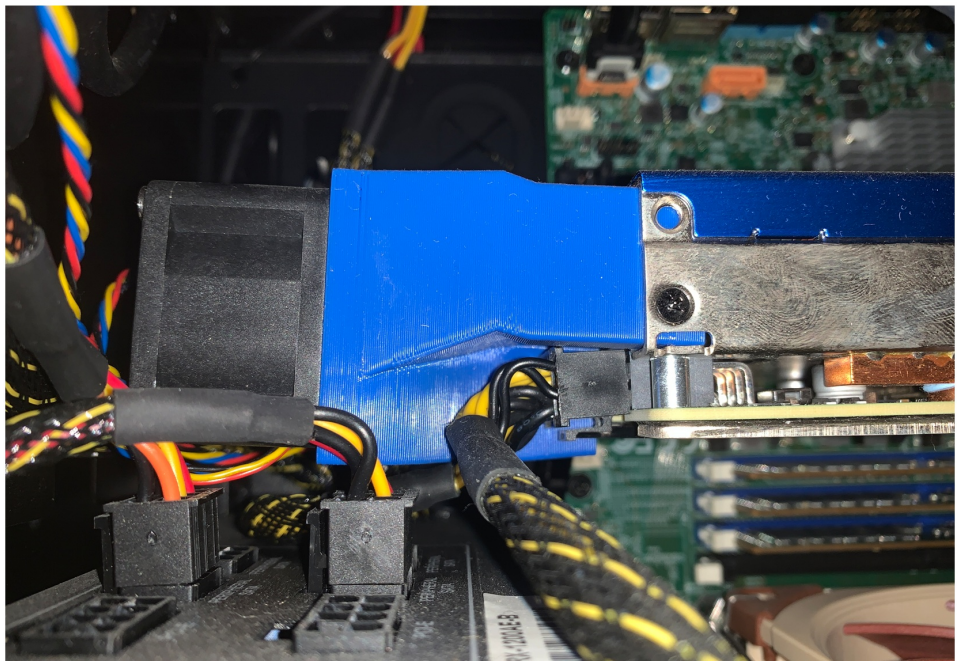


Fig. 1.1.12 [Compressed Air Cooling for the Xeon Phi Coprocessors]

HPMC 2026 by Bassem Jamaleddine

If the sound of the fan is annoying, then you can use liquid cooling for your MIC. However, this requires moving the MIC internal board to the Alpha cooling chassis. Figure 1.1.13 shows the Alpha Cool for the Xeon Phi



SAMPLE BOOK PREVIEW

Fig. 1.1.13 [Liquid Cooling for the Xeon Phi Coprocessors]

HPMC 2026 by Bassem Jamaleddine

SAMPLE BOOK PREVIEW

■ Cooling the GPU

The HPMC has a GTX TITAN X with 12GB that is liquid cooled as shown in Figure 1.1.14.

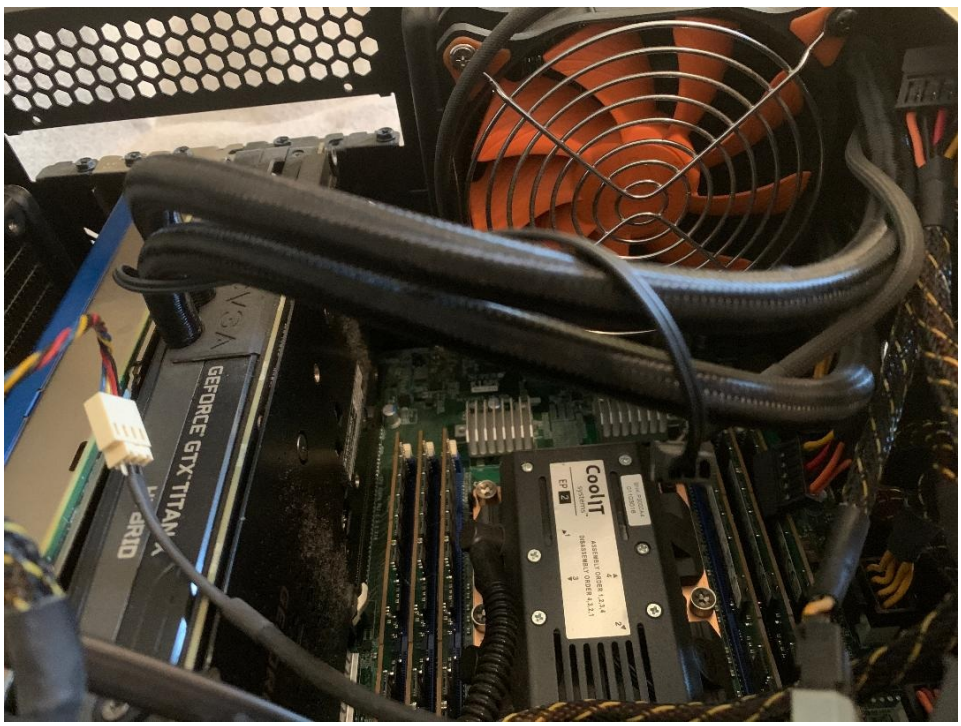
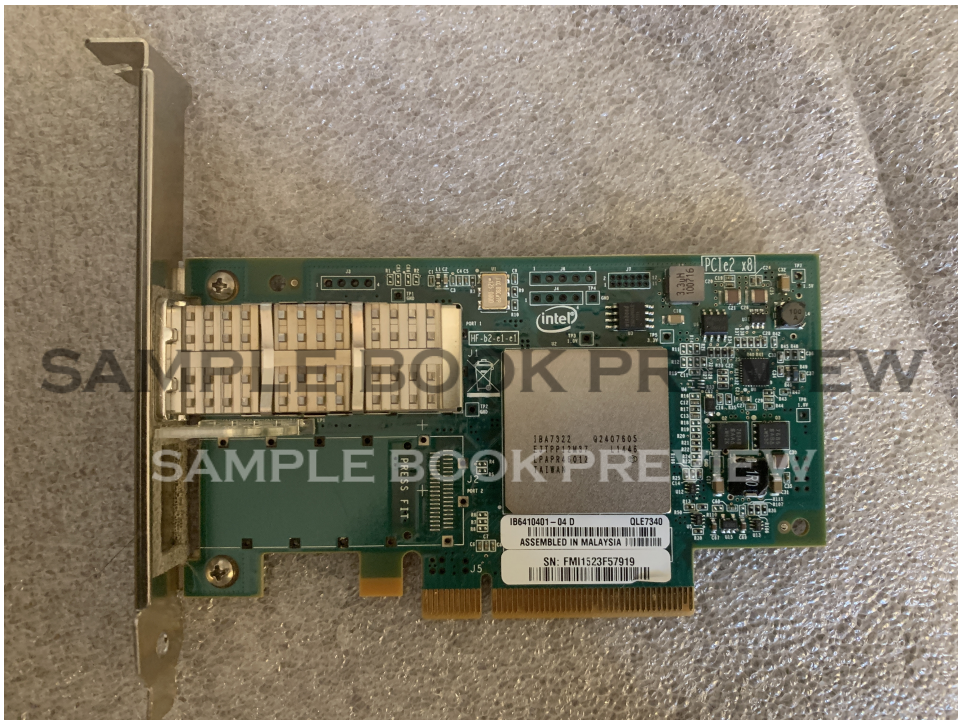


Fig. 1.1.14 [HPMC-KLIB Max's Inside View]

■ The SWITCH

The HPMC is equipped with an HCA so it can be connectted to the switch. By connecting HPMC calculators to the switch, you can have them participate in running programs over RDMA (Remote Direct Memory Architecture) using MPI (Message Passing Interface). Figure 1.1.16 shows the HCA. Figure 1.1.17 shows the swicth. Figure 1.1.18 shows connecting the HPMC to the switch.



SAMPLE BOOK PREVIEW
SAMPLE BOOK PREVIEW

Fig. 1.1.15 [HCA]

HPMC 2026 by Bassem Jamaleddine



SAMPLE BOOK PREVIEW
SAMPLE BOOK PREVIEW

Fig. 1.1.16 [Swicth]

HPMC 2026 by Bassem Jamaleddine

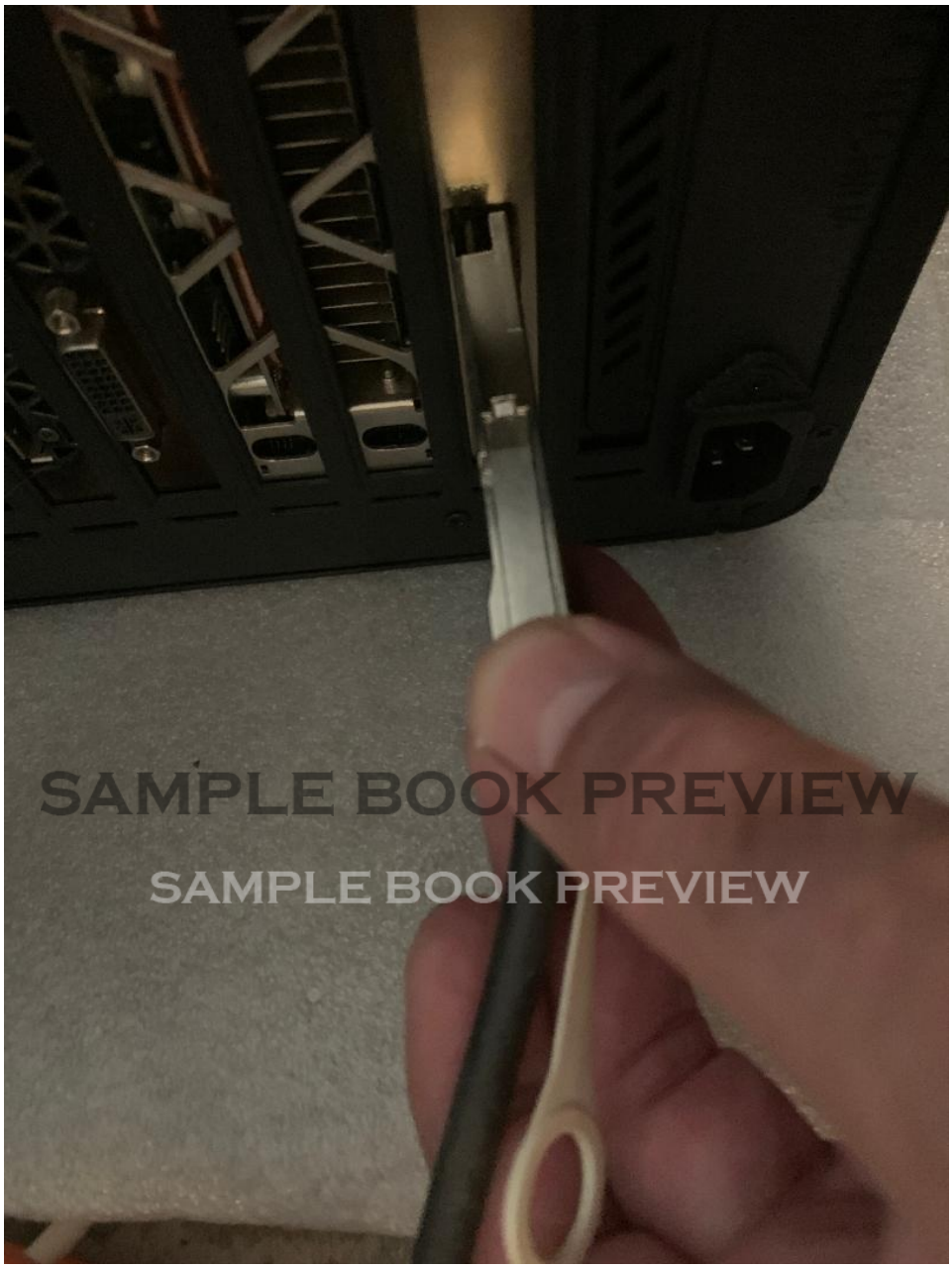

SAMPLE BOOK PREVIEW
SAMPLE BOOK PREVIEW

Fig. 1.1.17 [Connecting the HPMC to the switch]

HPMC 2026 by Bassem Jamaleddine

Fiber connectors connecting hpmc9 to the switch are shown in Figure 1.1.18. Notice that we have dual connectors for high availability. Having two connectors will not increase the performance. HPMC with PCIe x8 slot can use the HCA QLE7140 to connect to the switch using a copper cable QSFP to CX4. The cable has an attenuation that vary with the length, the shoter the better. With thin 30 AWG copper for 1 meter, 28 AWG for 3 meters, and thick 26 AWG for 6 meters. Figure 1.1.19 shows the HCA and its copper cable.

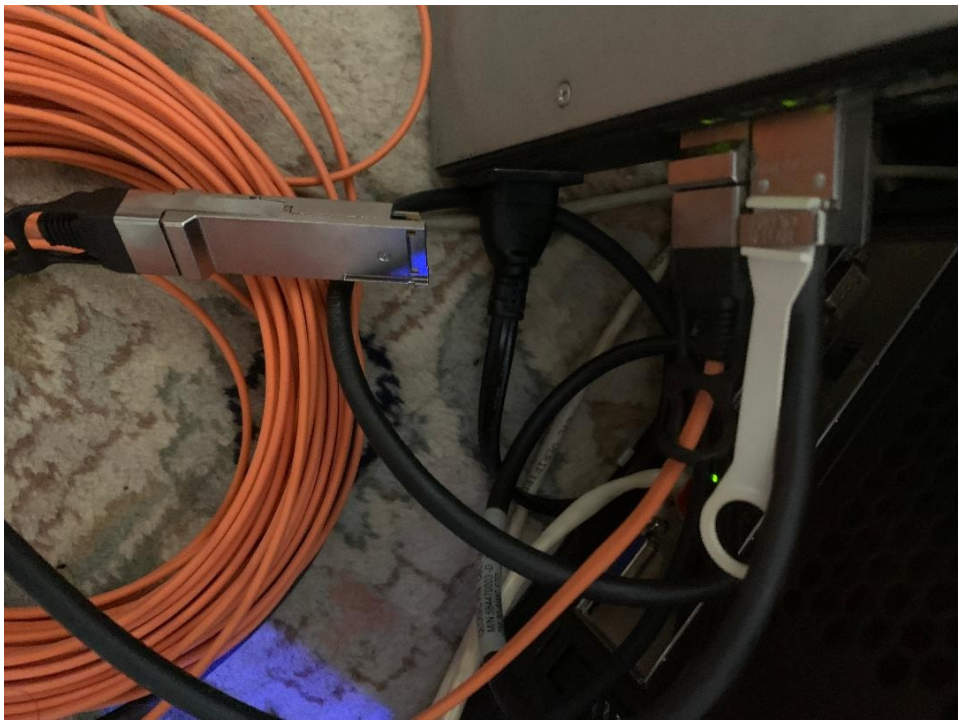


Fig. 1.1.18 [Fiber Connectors in Switch]

HPMC 2026 by Bassem Jamaleddine

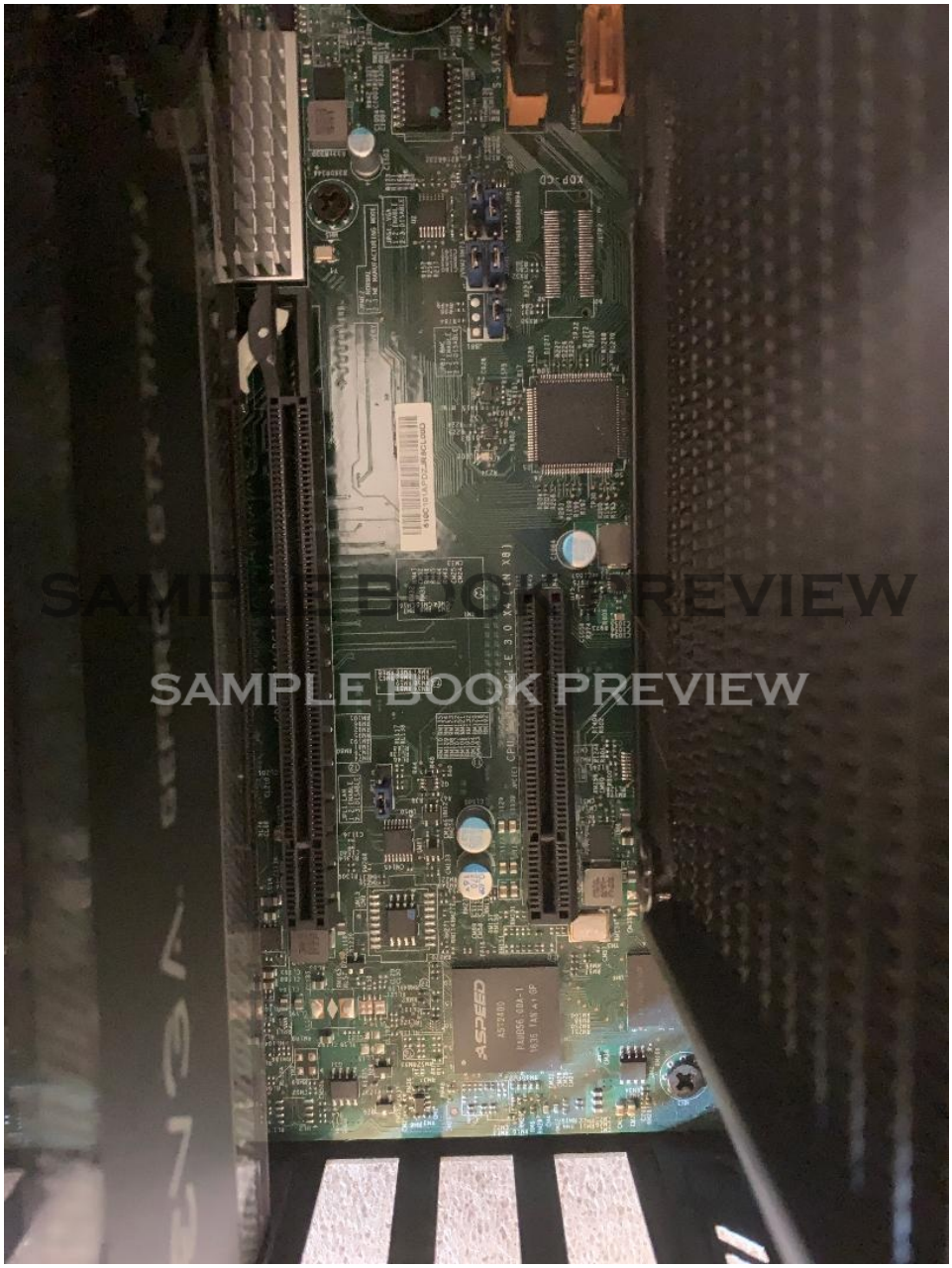
SAMPLE BOOK PREVIEW



Fig. 1.1.19 [QLOGIC QLE7140 Infiniband and Copper Cable QSFP to CX4]

HPMC 2026 by Bassem Jamaleddine

When selecting a board, make sure that there is enough room between the PCIe's. The MIC and the GPU are wide cards and each occupies the space of two slots on the board. Figure 1.1.20 shows the adequate space between the PCIe slots.



SAMPLE BOOK PREVIEW

Fig. 1.1.20 [Space between the PCIe's]

HPMC 2026 by Bassem Jamaleddine

To cool the HCA, you can use static air, pushing the air from the inside to the outside as shown in the following figure. Figure 1.1.21 shows two HCAs adapters and the static air cooling inside the hpmc9.



Fig. 1.1.21 [Cooling the HCA adapters]

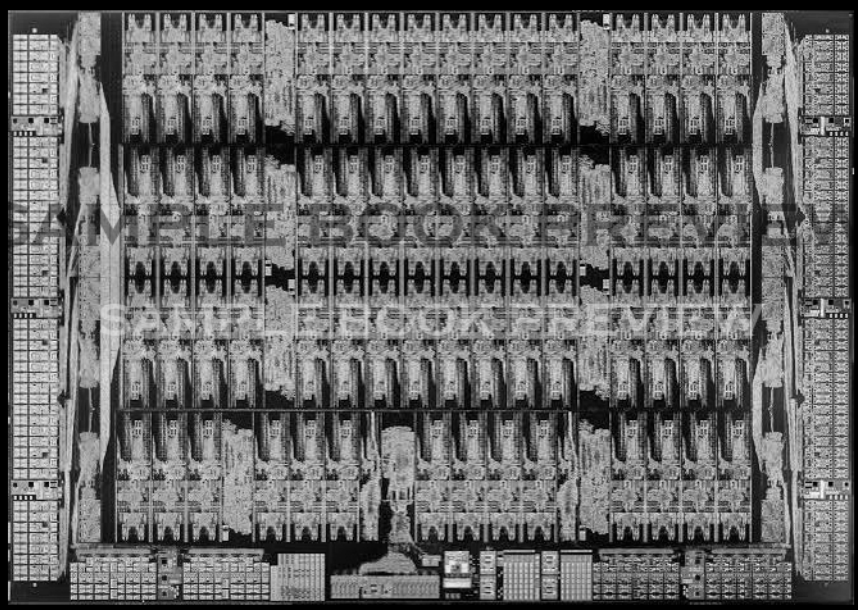
HPMC 2026 by Bassem Jamaleddine

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW


SAMPLE BOOK PREVIEW
SAMPLE BOOK PREVIEW

Chapter 2
Xeon Phi MIC

Xeon Phi MIC allows you to leverage x86 architecture (CPU with many cores) but allow for more compute throughput by dedicating much of the silicon to floating point ops. The MIC is cache coherent and it increases floating-point throughput. It also uses wide SIMD registers for more throughput. The MIC has fast GDDR5 memory integrated on the card. Later model, like the Knights Landing, features high-bandwidth on-package memory (MCDRAM) to reduce bottlenecks.

The MIC uses numerous low-power cores (60+) to deliver high parallel performance. Unlike GPUs, it supports standard x86 instruction sets, allowing the use of familiar languages like C++ and Fortran along with programming models such as OpenMP, MPI, and Pthreads.

The MIC can operate in three modes: Coprocessor/Offload (offloading tasks from the host CPU), Native (running as a standalone node with its own Linux OS), and Symmetric (using host and card together).

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

2-1 Configuring the Xeon Phi MIC Coprocessor

The HPMC is equipped with a Xeon Phi MIC coprocessor or many-core integrated architecture coprocessor, that is slotted in its PCIe bus. This coprocessor is a memory mapped device, and the motherboard hosting it must be capable to reserve the memory above 4GB for it own use.

Unlike legacy math coprocessor, the MIC is a device that runs its own Linux operating system (micro OS) and can be viewed as an embedded system that is plugged on the bus of the HPMC computer.

To bootstrap and system manage the MIC, we need to install on the host the "manycore processor software stack", which is the following package that contains the required modules: **mpss-modules-3.10.0-693.17.1.el7.x86_64-3.8.4-1.x86_64**. This package includes the following:

```
/etc/modprobe.d/mic.conf
/etc/sysconfig/modules/mic.modules
/etc/udev/rules.d/50-udev-mic.rules
/lib/modules/3.10.0-693.17.1.el7.x86_64/extra/mic.ko
```


```
/etc/modprobe.d/mic.conf
/etc/sysconfig/modules/mic.modules
/etc/udev/rules.d/50-udev-mic.rules
/lib/modules/3.10.0-693.17.1.el7.x86_64/extra/mic.ko
```


SAMPLE BOOK PREVIEW

Since the MIC card is a coprocessor that is made available on a host computer as an **embedded Linux system**, it needs to be booted as a computer. For this to happen:

- the MIC device driver must be loaded, hence to communicate and control the MIC hardware layer: lsmod | grep mic
- the MIC is recognized as a device to the host computer: ls /dev/mic0
- the MPSS software layer must be available on the host computer. If it has been installed then: which micinfo

To find information about your HPMC computer motherboard:

dmidecode -t 2

dmidecode -t memory

Let's have a little exercise with some commands to surface information about the MIC slotted in our HPMC.

14:23 root@HPMC7: ~ # **lspci | grep -i Phi**

```
14:23 root@HPMC7: ~ #  lspci | grep Phi
03:00.0 Co-processor: Intel Corporation Xeon Phi coprocessor SE10/7120 series (rev 20)
```

14:25 root@HPMC7: ~ # **cat /proc/iomem | grep 'PCI Bus'**

```
14:25 root@HPMC7: ~ #  cat /proc/iomem | grep 'PCI Bus'
000a0000-000bffff : PCI Bus 0000:00
  000c4000-000cbfff : PCI Bus 0000:00
d0000000-fbffbfff : PCI Bus 0000:00
  fa000000-fb0fffff : PCI Bus 0000:02
  fb400000-fb8fffff : PCI Bus 0000:01
  fb900000-fb9fffff : PCI Bus 0000:05
  fba00000-fbafffff : PCI Bus 0000:03
200000000000-3fffffffffff : PCI Bus 0000:00
  200000000000-2000000fffff : PCI Bus 0000:05
  3ff800000000-3ffbffffffff : PCI Bus 0000:03
  3ffc00000000-3ffc11ffffff : PCI Bus 0000:02
```

Here we got many PCI bus numbers, and we still need to know which one is used for the MIC. Let's look into /proc/iomem for the MIC, the address reported should be the same as the one shown above. This way we know the PCI slot.

```
14:26 root@HPMC7: ~ #  cat /proc/iomem | grep mic

14:26 root@HPMC7: ~ #  cat /proc/iomem | grep mic
      fba00000-fba1ffff : mic
      3ff800000000-3ffbfffffff : mic
```

So the slot is 03:00.0, same as reported in the first command. Let's get to where it expands in its own tree.

```
14:27 root@HPMC7: ~ #  lspci -s 03:00.0 -t

14:27 root@HPMC7: ~ #  lspci -s 03:00.0 -t
-+-[0000:03]---00.0
 \-[0000:00]-
```

So we will suffix 0000:03 with :00.0 and grep the /proc/iomem

```
14:28 root@HPMC7: ~ #  cat /proc/iomem | grep '0000:03:00.0'

14:28 root@HPMC7: ~ #  cat /proc/iomem | grep '0000:03:00.0'
    fba00000-fba1ffff : 0000:03:00.0
    3ff800000000-3ffbfffffff : 0000:03:00.0
```

SAMPLE BOOK PREVIEW

To finally disclose the MIC memory I/O remapping, take the prefix 3ff800000000 used for the PCIe slot 0000:03:00.0 and grep /proc/vmallocinfo.

```
14:29 root@HPMC7: ~ #  cat /proc/vmallocinfo | grep '3ff800000000'

14:29 root@HPMC7: ~ #  cat /proc/vmallocinfo | grep '3ff800000000'
0xffffc90040000000-0xffffc90440001000 17179873280 ioremap_work+0x24/0xd0 [mic] phys=3ff800000000 ioremap
```

This command reveals the mapping of virtual address space region created by ioremap (to map the MIC as a PCI device to the I/O memory region). The MIC is a hardware device slotted in the mainboard PCI bus that requires 4G of memory mapped IO. The kernel reserves the memory area by mapping the I/O memory region so we can interact with the device by simple write/read in that area.

```
02:30 root@HPMC7: ~ #  lspci -s 03:00.0 -v

02:30 root@HPMC7: ~ #  lspci -s 03:00.0 -v
03:00.0 Co-processor: Intel Corporation Xeon Phi coprocessor SE10/7120 series (rev 20)
        Subsystem: Intel Corporation Device 7d95
        Physical Slot: 1
        Flags: bus master, fast devsel, latency 0, IRQ 28, NUMA node 0
        Memory at 3ff800000000 (64-bit, prefetchable) [size=16G]
        Memory at fba00000 (64-bit, non-prefetchable) [size=128K]
        Capabilities: [44] Power Management version 3
        Capabilities: [4c] Express Endpoint, MSI 00
        Capabilities: [88] MSI: Enable- Count=1/16 Maskable- 64bit+
        Capabilities: [98] MSI-X: Enable+ Count=16 Masked-
        Capabilities: [100] Advanced Error Reporting
        Kernel driver in use: mic
```

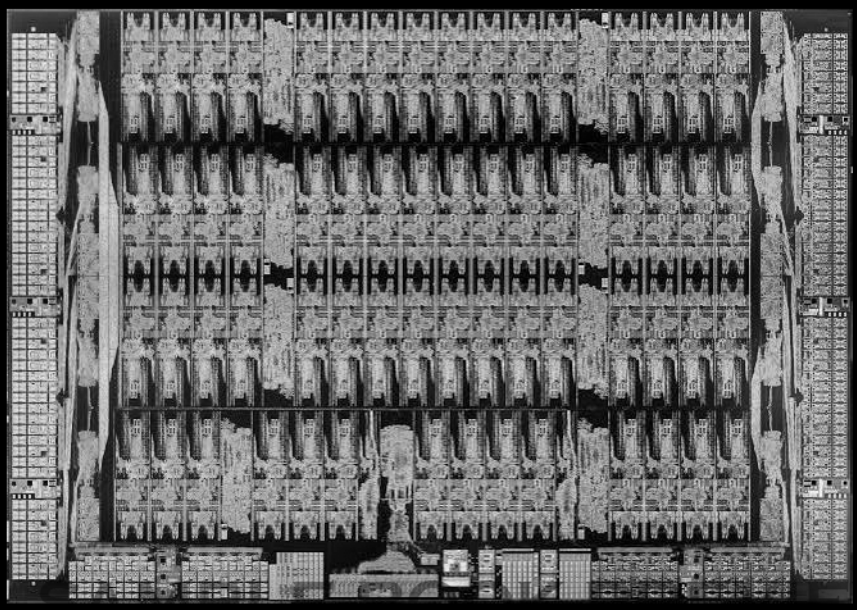
■ Adding the MIC to the Host Network Services

Configure your system network layers so that the Xeon Phi MIC devices can be daisy chained together and be part of your system. The MIC card is a high performance computer by itself that can be booted, configured, and made available for offloading math intensive computations. To have the MIC configured on your system, create a bridge to combine the multiple network adapters: your MIC cards and the different adapters can then talk to each others as if directly connected to a normal network switch.

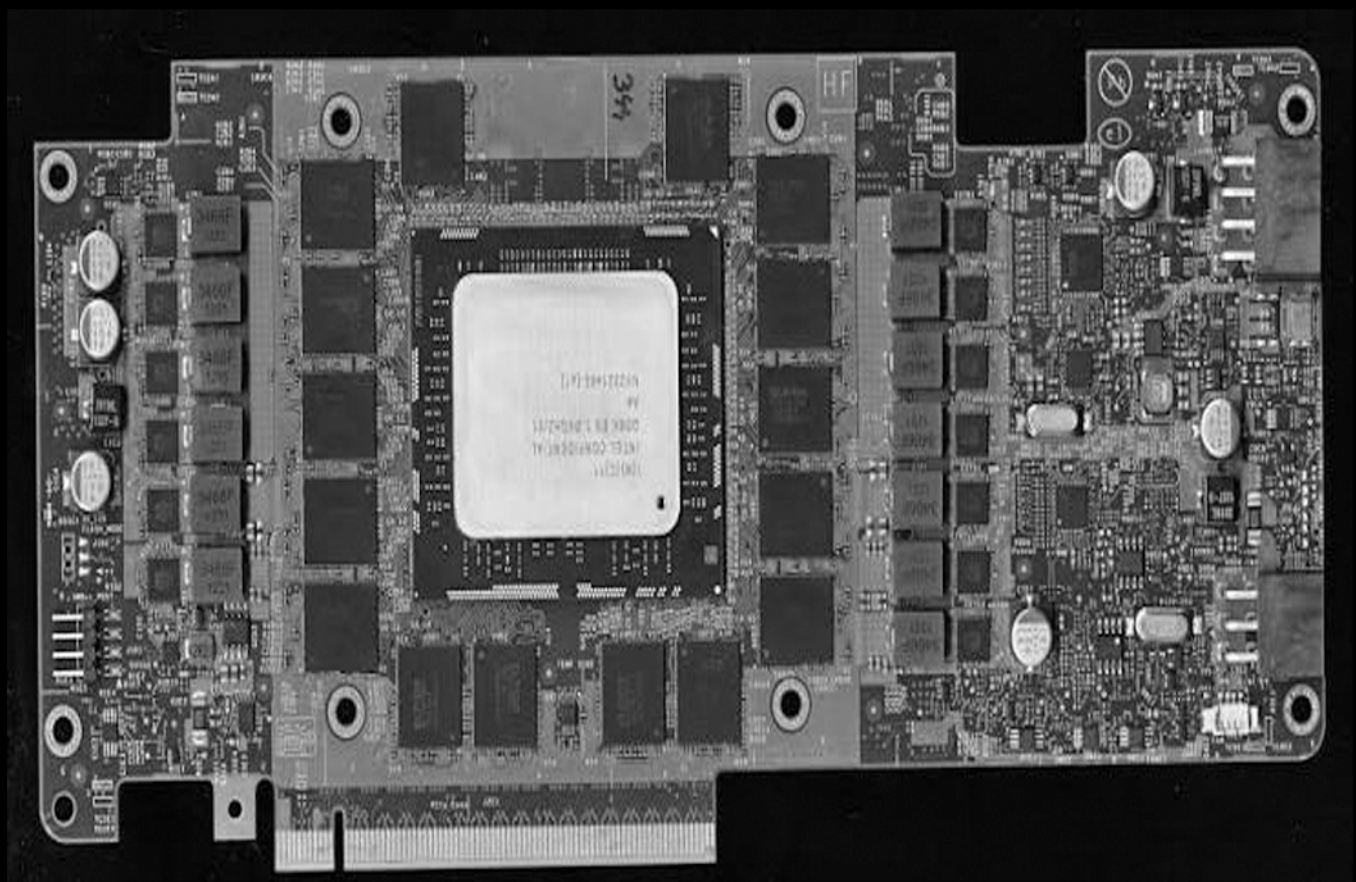
In the directory /**etc**/**sysconfig**/**network-scripts**, create a bridge by editing the file **ifcfg-br0**

```
DEVICE=br0
```

SAMPLE BOOK PREVIEW



SAMPLE BOOK PREVIEW



Chapter 3
Computer Organisation of the MIC

Intel took one P5 CPU, blasted its frequency to the upper 1GHz, threaded it with four logical processing units, wired it to a VPU with 128x 512-bits registers (hence 32x 512-bits registers per thread), added eight master ALUs each containing two SP and one DP ALU with independent pipelines (DP pipeline and SP pipeline), and replicated it 61 times. This chapter looks at the computer organisation of the MIC from the system point of view. This means we will be looking into the Linux system directories to derive the layout of the MIC memory, caches, threads, and interrupts. We will develop some commands to exploit the MIC native code. These commands are basic shell scripts that call the gdb debugger on some source code.

In addition, this chapter addresses the MIC vector registers, vector instruction, intrinsic operations, pipelined operations, etc.

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

3-1 MIC Cores Threads and Cache Organization

Previously, in the Chapter "Configuring the Xeon Phi MIC Coprocessor", we showed the MIC installation from the system point of view: memory mapped IO, device drivers and device mapping of the MIC as a card slotted in the PCI bus of our HPMC. It is time to look at its architecture from an administrative point of view, so our focus is on how the MIC cores and their logical processors are laid out (on the die) to interact with memory.

■ MIC Cores and Cache Memory Organization

Since all cores are identical, let's look at the memory caching of one of the cpu, say cpu59:

(uos/HPMC7) 18:13 root@mic0: ~ # **cat /sys/devices/system/cpu/cpu59/cache/index0/level**

1

(uos/HPMC7) 18:13 root@mic0: ~ # **cat /sys/devices/system/cpu/cpu59/cache/index0/size**

32K

(uos/HPMC7) 18:13 root@mic0: ~ # **cat /sys/devices/system/cpu/cpu59/cache/index1/level**

1

(uos/HPMC7) 18:13 root@mic0: ~ # **cat /sys/devices/system/cpu/cpu59/cache/index1/size**

32K

(uos/HPMC7) 18:13 root@mic0: ~ # **cat /sys/devices/system/cpu/cpu59/cache/index2/level**

2

(uos/HPMC7) 18:14 root@mic0: ~ # **cat /sys/devices/system/cpu/cpu59/cache/index2/size**

512K

(uos/HPMC7) 18:14 root@mic0: ~ # **cat /sys/devices/system/cpu/cpu59/cache/index2/shared_cpu_list**

57-60

SAMPLE BOOK PREVIEW

(uos/HPMC7) 18:20 root@mic0: ~ # **cat /sys/devices/system/cpu/cpu59/cache/index1/shared_cpu_list**

57-60

(uos/HPMC7) 18:20 root@mic0: ~ # **cat /sys/devices/system/cpu/cpu59/cache/index0/shared_cpu_list**

57-60

SAMPLE BOOK PREVIEW

Here cpu59 has 512K of L2 cache that is shared with cpu57, cpu58, cpu59, and cpu60. All four of these cpus pertain to core #14. Core #14 has four threads represented by cpu57, cpu58, cpu59, and cpu60. All four threads are represented as logical processors (LPs) that share the L2 cache 512K. They also share the L1 cache 32K. This means that each core caches is shared by the threads it represents.

Let's take a look at core #61, that should be the core that provides the last thread or thread number 243, that is 61*4 minus one since we start counting at 0th in the mapping.

(uos/HPMC7) 22:10 root@mic0: ~ # **cat /sys/devices/system/cpu/cpu243/cache/index0/shared_cpu_list**

0,241-243
Core #61 represents the four logical processors (or hardware threads) represented by cpu241, cpu242, cpu243, and cpu0.

Each core is a multithreaded processor with superscalar execution units. In each core there are four threads that appear like four processors sharing the same cache. These four logical processors (LP0 to LP4) are mapped to their corresponding directory in /sys/devices/system/cpu/cpu*

(uos/HPMC) 10:44 root@mic0: ~ # **ls -ld /sys/devices/system/cpu/cpu[0-9]* | wc -l**

244

To view the organization of the MIC cores and their corresponding threads, you can list the system CPU directory on the MIC by using the following command.

(uos/HPMC) 10:47 root@mic0: ~ # **ls -ld /sys/devices/system/cpu/cpu[0-9]*/***

```
drwxr-xr-x    5 root     root             0 Mar 21 10:44 /sys/devices/system/cpu/cpu0/cache
drwxr-xr-x    3 root     root             0 Mar 15 12:50 /sys/devices/system/cpu/cpu0/cpufreq
drwxr-xr-x    5 root     root             0 Mar 21 10:44 /sys/devices/system/cpu/cpu0/cpuidle
-r--------    1 root     root          4096 Mar 20  2018 /sys/devices/system/cpu/cpu0/crash_notes
drwxr-xr-x    2 root     root             0 Mar 21 10:44 /sys/devices/system/cpu/cpu0/topology
drwxr-xr-x    5 root     root             0 Mar 21 10:44 /sys/devices/system/cpu/cpu1/cache
lrwxrwxrwx    1 root     root             0 Mar 15 12:50 /sys/devices/system/cpu/cpu1/cpufreq -> ../cpu0/cpufreq
drwxr-xr-x    5 root     root             0 Mar 21 10:44 /sys/devices/system/cpu/cpu1/cpuidle
-r--------    1 root     root          4096 Mar 20  2018 /sys/devices/system/cpu/cpu1/crash_notes
-rw-r--r--    1 root     root          4096 Mar 21 10:44 /sys/devices/system/cpu/cpu1/online
drwxr-xr-x    2 root     root             0 Mar 21 10:44 /sys/devices/system/cpu/cpu1/topology
drwxr-xr-x    5 root     root             0 Mar 21 10:44 /sys/devices/system/cpu/cpu10/cache
...
```

You can also display the lengthy output of the cpuinfo.

(uos/HPMC) 13:56 root@mic0: ~ # **cat /proc/cpuinfo**

In general, in each core there are four threads, we will represent them as four logical processors (LP0 to LP4) as shown in the following schematic representation.

```
 +----------------+--------+-------------+
 | LP0 - thread   |        |             |
 | LP1 - thread   | CACHE  |  Memory     |
 | LP2 - thread   |        | Controller  +---> interconnect ----> RAM
 | LP3 - thread   |        |             |
 +----------------+--------+-------------+
```

Each core has a 32KB L1 data cache, a 32KB L1 instruction cache,and a 512KB L2 cache. The L2 caches of all cores are interconnected with each other and the memory controllers via a bidirectional ring bus, effectively creating a shared last-level cache of up to 32MB. The design of

each core includes a short in-order pipeline. There is no latency in executing scalar operations and low latency in executing vector operations. Due to the short in–order pipeline, the overhead for branch misprediction is low.

Each core includes a Vector Processing Unit (VPU). Each vector unit contains 128 512-bit vector registers divided over the four threads, thus 32 512-bit vector registers per thread. As we have seen in the previous chapter, the gdb showed the 32 512-bit registers kmm0 to kmm31.

To support the new vector processing model, a new 512-bit SIMD ISA was introduced. The VPU is a key feature of the Intel MIC Architecture-based cores. Fully utilizing the vector unit is critical for the bestIntel Xeon Phi coprocessor performance. It is important to note that Intel MIC Architecture cores do not support other SIMD ISAs

The biggest change to the cores is adding support for 64 bit instructions, as well as adding a new execution unit called the VPU. This VPU (Vector Processing Unit) has its own 512-bit wide SIMD instruction set, integer support, Fused Multiply/Add, and other advanced features that are more commonly found in GPU's. The VPU is the result of Intel's work with Larrabee, the precursor to Knights Corner. With the VPU, each core has 4 execution units (VPU, FXU, and 2 x Integer units). This allows the cores to support 4-way multi-threading; in practice, 2 threads are most common as 2 execution units are usually tied up calculating memory addresses.

Cloning the whole core hundred of times to create a homogeneous manycore architecture, hence to gain parallelism to be exploited by the codes designed to run on such architecture. All cores are connected into a ring network so that all their thread appears as processors (P0 .. P243), where one or many memory controllers interconnet them to the memory (managed by the IO edge interrupts).

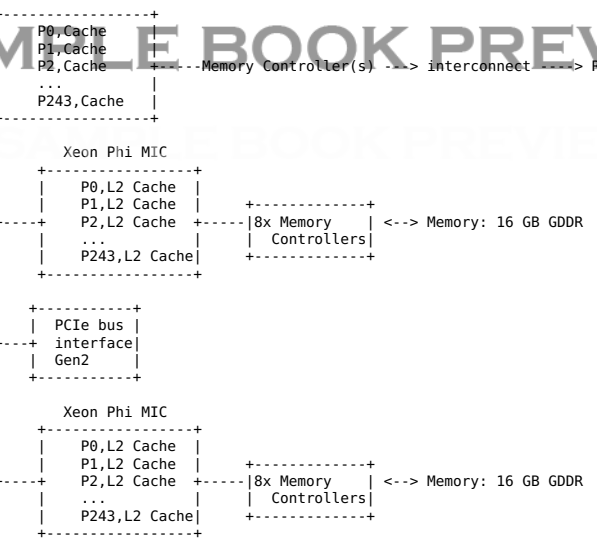
16GB are shared by all MIC cores. Login to the MIC and print the available memory in /proc/meminfo.

```
(uos/HPMC7) 17:59 root@mic0: ~ #  cat /proc/meminfo | grep -i memtotal


MemTotal:       15885480 kB
```

There are 16 GB of memory that are accessible by all processors via eight memory controllers with two GDDR5 channels running at 5.5 GT/s; so the aggregated memory bandwith is 352 BS/s (8 memory controllers x 2 channels x 5.5 (GT/s) x 4 bytes/transfer) The system interface of the chip supports PCIe2 x 16 protocols with 256-byte packets.

SAMPLE BOOK PREVIEW



The 61 cores have 244 logical processors that are represented on the MIC with numbers 0 .. 243 as this can be shown in the MIC's /proc/cpuinfo.

```
(uos/HPMC7) 11:35 root@mic0: /tmp #  cat /proc/cpuinfo | grep -i -E "initial apicid
  processor" | head


processor       : 0
initial apicid  : 240
processor       : 1
initial apicid  : 0
processor       : 2
initial apicid  : 1
processor       : 3
initial apicid  : 2
processor       : 4
initial apicid  : 3
```

```
(uos/HPMC7) 11:36 root@mic0: /tmp #  cat /proc/cpuinfo | grep -i -E "initial apicid
  processor" | tail -n 4


processor       : 242
initial apicid  : 242
processor       : 243
initial apicid  : 243
```

The HPMC hpmc7 is hosting the MIC mic0 that is inserted in its PCIe2 x16 bus. hpmc7 mainboard has a processor 7250 Xeon Phi Knights Landing, it has 68 cores representing 272 threads. hpmc7 is not a MIC coprocessor.

```
12:09 root@HPMC7: ~ #  grep -m 1 'model name' /proc/cpuinfo
```

```
model name      : Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz
```

12:10 root@HPMC7: ~ # **cat /proc/cpuinfo | head -n 30**

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 87
model name      : Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz
stepping        : 1
microcode       : 0x1b0
cpu MHz         : 1055.687
cache size      : 1024 KB
physical id     : 0
siblings        : 272
core id         : 0
cpu cores       : 68
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov ...
bogomips        : 2799.86
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 87
```

The following is the full list of flags for the Knights Landing (KLN) processor:

```
fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
```

The Knights Landing processor has AVX-512F (Advanced Vector Extensions 512 Foundation) instruction set, while the Knights Corner does not. The MIC coprocessor instruction set is IMCI (Initial Many-Core Instruction set). Regardless of the marketing lexicon, we won't be discussing the difference between AVX and IMCI. However, as we will see in some programming example, the MIC has extensive vector instructions executed by the VPUs (Vector Processing UNITs).

SAMPLE BOOK PREVIEW

Looking at HPMC hpmc7 that has a KLN processor, the AVX signature is shown in the /proc/cpuinfo.

12:08 root@HPMC7: ~ # **grep -o 'avx[^]*' /proc/cpuinfo | head**

SAMPLE BOOK PREVIEW

```
avx
avx2
avx512f
avx512pf
avx512er
avx512cd
avx
avx2
avx512f
avx512pf
```

hpmc7 has 272 threads represented by cpu0 .. cpu273. Each thread has a cache 1-MB or 1024-KB.

12:05 root@HPMC7: ~ # **grep "cache size" < /proc/cpuinfo | wc -l**

```
  272
```

The cache size of the hpmc7 processor is shown below, with 272x 1-MB caches, one per each thread.

12:06 root@HPMC7: ~ # **grep "cache size" < /proc/cpuinfo | head**

```
cache size      : 1024 KB
cache size      : 1024 KB
cache size      : 1024 KB
cache size      : 1024 KB
cache size      : 1024 KB
cache size      : 1024 KB
cache size      : 1024 KB
cache size      : 1024 KB
cache size      : 1024 KB
cache size      : 1024 KB
...
```

18:19 root@HPMC7: ~ # **cat /sys/devices/system/cpu/cpu2/cache/index2/level**

2

18:21 root@HPMC7: ~ # **cat /sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_list**

0,68,136,204

The mapping is for each core to its logical processor is done in modulo 67, wrapped around on the 67th count.

SAMPLE BOOK PREVIEW

3-2 MIC Superscalar Vector Processor a Programmer View

In this chapter, we will take a look at the MIC internal processing from a programmer point of view by disassembling a program and address some of the instructions executed by the MIC to prove that the MIC is a superscalar processor, as well a vector processor. Whether the program is written in C, CPP, or FORTRAN, our focus is to look at its MIC native code. We will write four simple shell scripts to filter the output of the gdb command when run on some compiled MIC native code.

The discussion will procede from bottom to top: first we will run the gdb to reveal the vector registers and the masking registers as reported by the debugger, second we will disassemble the program mic-gigaflops.cpp and look into its corresponding assembly code. By explaining some of these assembly instructions, the reader will gain some understanding of the MIC internal vector processor units. Each core on the MIC can execute instructions to operate sequentially on large 512-bit wide one-dimensional arrays of data called vectors. The vector processor is integrated as a hardware component called a vector processing unit (VPU). This is in contrast to scalar processors, whose instructions operate on single data items only, and in contrast to some of those same scalar processors having additional single instruction, multiple data (SIMD).

Since each core on the MIC has a vector processing unit (VPU), therefore, for the 61 cores on the MIC there are 61 VPUs. The VPU has two independent pipelines (known as the U and V pipelines) allowing the core to have two independent instructions started in the execution stage at the same clock cycle. Since each core has four threads, then *two or more* of these threads can fill the VPU pipelines for the maximum performance. Therefore the cores on the MIC are superscalar processors that can execute multiple operations in a single cycle.

SAMPLE BOOK PREVIEW

In general we say that the MIC instructions can be pipelined at a throughput rate of one vector instructions per cycle. Having two pipelines per core; therefore, each core is a 2-wide processor that can execute two instructions per cycle, one on the U-pipe and the other on the V-pipe. It also contains an x87 unit to perform floating-point instructions when needed. The cores execute vector instructions, as each core has vector units that consist of 32x 512-bit vector registers and 8 mask registers to allow predicated execution on the vector elements. Support for Scatter/Gather vector memory instructions, floating-point operations per IEEE 745, support for transcendental instructions for exp, log, recip, and sqrt functions in hardware.

■ GDB Scripts for the MIC

We are only concerned with the architecture of the MIC that is reflected by writing a piece of code to be executed on it. Rather then to look into the MIC hardware organization and its instruction set, we will look into few of its intructions to do the basic arithmetics and system I/O.

Let's look into the assembly code of the program mic-gigaflops.cpp from Chapter "Performance-Programming-for-MIC" section "Measuring Practical Gigaflops to the MIC Specification Gigaflops". To do that, we will simply compile the program, then disassemble it using gdb. Once disassembled, we will look into the instructions. It is simpler and more exciting to approach this discussion from the bottom up: from the assembly source code, to how it is executed on the MIC. So our approach is to understand to MIC architecture from the instruction set.

Command script	Example	Output file
gdbmic-disas	gdbmic-disas mic-gigaflops	mic-gigaflops_xphi_Gg.MIC-ASM
gdbmic-reg	gdbmic-reg mic-gigaflops	mic-gigaflops_xphi_Gg.MIC-REG
gdbmic-run	gdbmic-run mic-gigaflops	mic-gigaflops_xphi_Gg.MIC-RUN
gdbmic-step	gdbmic-step mic-gigaflops 300	mic-gigaflops_xphi_Gg.MIC-STEP
objdump-mic	objdump-mic mic-gigaflops	mic-gigaflops_xphi_Gg.ASMi

Tab. 3.2.1 [gdb MIC Scripts]

Looking at table 3.2.1, each of the commands in the first column is the name of simple shell scripts

SAMPLE BOOK PREVIEW

3-3 MIC Vector Operations a Programmer View

When writing programs for the MIC, you do not need to be concerned with its internal architecture. However, it is useful to understand how the vector operations are being performed, in particular to clarify the fact of what makes your HPMC (equipped with the MIC) a superscalar vector processor, in other term: a supercomputer. In this chapter, we will simplify the view of the vector processing over the vector registers: the vector will be schematically presented from a programmer point of view. First, we will go over a simple program, vector512.cpp, that we will disassemble, then draw the vector register by mapping it through four lanes. This allows us to clarify how the vector instructions are being executed on the MIC. Next, we will show various programs that uses the intrinsic operations to exploit the MIC vector registers when performing fast addition and multiplication on data arrays. This subchapter concludes on the subject of data casting, and why mixing data types should be avoided.

■ Vector Operations: vector512.cpp

Consider the following program vector512.cpp that contains various intrinsic operations to manipulate a list of sixteen float data elements saved in a vector.

-LIS- Listing. 3.3.1 [Vector Operations on the MIC: vector512.cpp]

SAMPLE BOOK PREVIEW

```
1.    #define MICVEC_DEFINE_OUTPUT_OPERATORS
2.    #include <iostream>
3.    #include <micvec.h>
4.
5.    int main() {
6.        _MM_PERM_ENUM p32;
7.        _MM_SWIZZLE_ENUM s32;
8.
9.        __declspec(align(64)) Is32vec16
      inputData(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);
10.       __declspec(align(64)) Is32vec16 outputData;
11.       std::cout << "input = " << inputData;
12.
13.       // swizzle input data and print
14.       std::cout << "\nswizzling with pattern 'cdab' \n" <<
      inputData.cdab();
15.       std::cout << "\nswizzling with pattern 'aaaa' \n" <<
      inputData.aaaa();
16.       std::cout << "\nswizzling with pattern 'bbbb' \n" <<
      inputData.bbbb();
17.       std::cout << "\nswizzling with pattern 'cccc' \n" <<
      inputData.cccc();
18.       std::cout << "\nswizzling with pattern 'dddd' \n" <<
      inputData.dddd();
19.       std::cout << "\nswizzling with pattern 'cdab' // ghef cdab - Swap
      pairs \n" << inputData.cdab();
20.       std::cout << "\nswizzling with pattern 'dacb' \n" <<
      inputData.dacb();
21.
22.       // shuffle input data
23.       std::cout << "\nInter lane shuffle data for pattern 'aaaa' \n";
24.       p32 = _MM_PERM_AAAA;
25.       //shuffle intra lane data
26.       outputData = Is32vec16(_mm512_shuffle_epi32(__m512i(inputData),
      p32));
27.       std::cout << outputData << "\n";
28.
29.       // shuffle input data
30.       std::cout << "Inter lane shuffle data for pattern 'dada' \n";
31.       p32 = _MM_PERM_DADA;
32.       //shuffle intra lane data
33.       outputData = Is32vec16(_mm512_shuffle_epi32(__m512i(inputData),
      p32));
34.       std::cout << outputData << "\n";
35.
36.       // Permute the four 128-bit elements of v2 according to indexes in
      'perm'.
37.       std::cout << "Inter lane permute data for pattern 'aabc' // Permute
      the four 128-bit elements \n";
38.       p32 = _MM_PERM_AABC;
```

```
39.       outputData =
      Is32vec16(_mm512_permute4f128_epi32(__m512i(inputData), p32));
40.       std::cout << outputData << "\n";
41.
42.       std::cout << "Inter lane permute data for pattern 'dacb' \n";
43.       p32 = _MM_PERM_DACB;
44.       //shuffle inter lane data
45.       outputData =
      Is32vec16(_mm512_permute4f128_epi32(__m512i(inputData), p32));
46.       std::cout << outputData << "\n";
47.
48.       std::cout << "Inter lane for register swizzle NONE \n";
49.       s32 = _MM_SWIZ_REG_NONE;
50.       outputData = Is32vec16(_mm512_swizzle_epi64(__m512i(inputData),
      s32));
51.       std::cout << outputData << "\n";
52.
53.       std::cout << "Inter lane for register swizzle for pattern 'dcba'
      \n";
54.       s32 = _MM_SWIZ_REG_CDAB;
55.       outputData = Is32vec16(_mm512_swizzle_epi64(__m512i(inputData),
      s32));
56.       std::cout << outputData << "\n";
57.
58.       std::cout << "Inter lane for register swizzle for pattern 'cdab' //
      ghef cdab - Swap pairs \n";
59.       s32 = _MM_SWIZ_REG_CDAB; //  /* ghef cdab - Swap pairs */
60.       outputData = Is32vec16(_mm512_swizzle_epi64(__m512i(inputData),
      s32));
61.       std::cout << outputData << "\n";
62.
63.       std::cout << "Inter lane for register swizzle for pattern 'dacb'
      CROSS PRODUCT // hegf dacb - cross-product \n";
64.       s32 = _MM_SWIZ_REG_DACB; //  /* hegf dacb - cross-product */
65.       outputData = Is32vec16(_mm512_swizzle_epi64(__m512i(inputData),
      s32));
66.       std::cout << outputData << "\n";
67.
68.       std::cout << "Inter lane for register swizzle for pattern 'badc' //
      fehg badc - Swap with two-away \n";
69.       s32 = _MM_SWIZ_REG_BADC; //  /* fehg badc - Swap with two-away */
70.       outputData = Is32vec16(_mm512_swizzle_epi64(__m512i(inputData),
      s32));
71.       std::cout << outputData << "\n";
72.
73.       std::cout << "Inter lane for register swizzle for pattern 'aaaa' //
      eeee aaaa - broadcast a element \n";
74.       s32 = _MM_SWIZ_REG_AAAA; //  /* eeee aaaa - broadcast a element */
75.       outputData = Is32vec16(_mm512_swizzle_epi64(__m512i(inputData),
      s32));
76.       std::cout << outputData << "\n";
77.    }
```

SAMPLE BOOK PREVIEW

HPMC 2026

The output of the program is shown in the following figure.

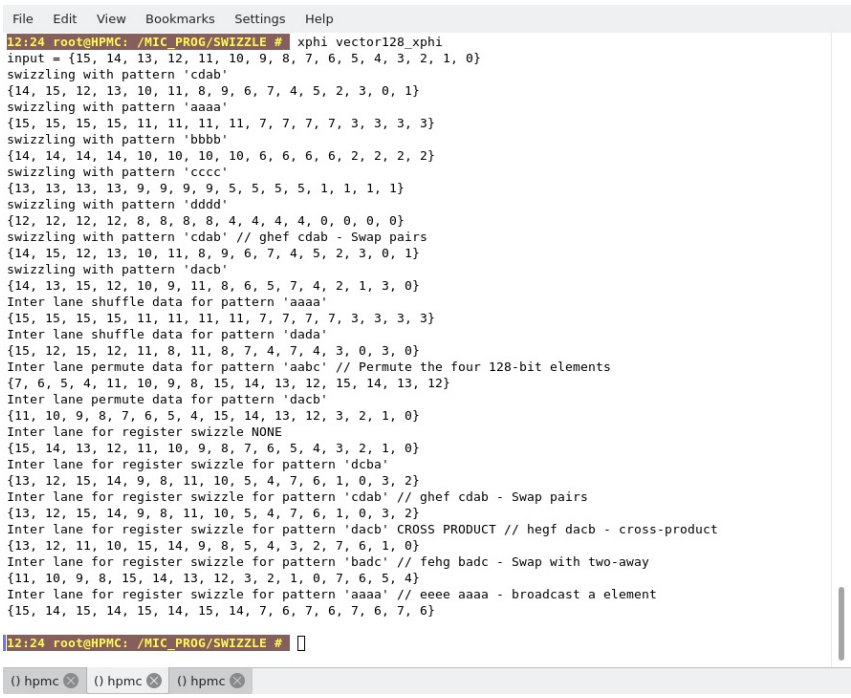


Fig. 3.3.1 [Vector operations running the program vector512.cpp]

SAMPLE BOOK PREVIEW

Let's disassemble the program:

SAMPLE BOOK PREVIEW

13:06 root@HPMC: /MIC_PROG/SWIZZLE # **gdbmic-disas vector512**

```
vector512
vector512.cpp
vector512_xphi_Gg
vector512_xphi_Gg.MIC-ASM
vector512_xphi_Gg                                    100%   89KB   9.2MB/s   00:00
vector512_xphi_Gg.MIC-ASM                            100%   73KB   8.6MB/s   00:00
```

We can grep for the instructions where two vector registers are being used:

13:06 root@HPMC: /MIC_PROG/SWIZZLE # **grep -o '.*zmm.*zmm.*' vector512_xphi_Gg.MIC-ASM**

```
vmovdqa32 zmm1,zmm0{cdab}
vmovdqa32 zmm1,zmm0{aaaa}
vmovdqa32 zmm1,zmm0{bbbb}
vmovdqa32 zmm1,zmm0{cccc}
vmovdqa32 zmm1,zmm0{dddd}
vmovdqa32 zmm1,zmm0{cdab}
vmovdqa32 zmm1,zmm0{dacb}
vmovdqa64 zmm1,zmm0{cdab}
vmovdqa64 zmm1,zmm0{cdab}
vmovdqa64 zmm1,zmm0{dacb}
vmovdqa64 zmm1,zmm0{badc}
vmovdqa64 zmm1,zmm0{aaaa}
```

The instruction **vmovdqa32** moves int32 vector result of the swizzle/broadcast/conversion process on int32 vector zmm0 into int32 vector zmm1.
The instruction **vmovdqa64** moves int64 vector result of the swizzle/broadcast/conversion process on int64 vector zmm0 into int64 vector zmm1. The swizzle pattern is written between the curly braces.

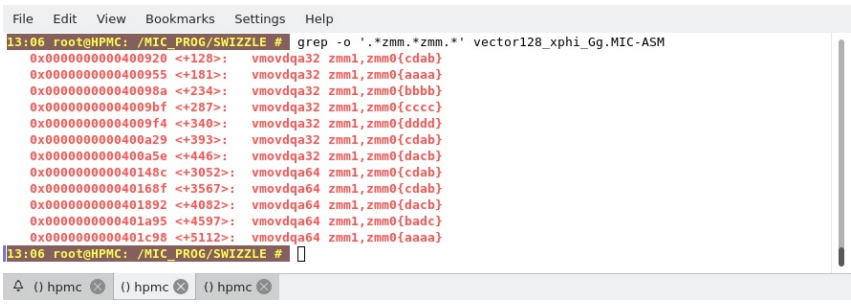


Fig. 3.3.2 [Vector Operations Swizzle vector512_assembly_swizzle.png]

HPMC 2026

The program vector512.cpp used symbols and functions defined in include/zmmintrin.h and include/immintrin.h. Both include files have been included with the statement

```
#include <micvec.h>
```

■ Vector Operations on Vector Registers with Four Lanes

The Intel Xeon Phi MIC has 61 cores, with each core has a VPU, Vector Processing Unit, that provides fine grained data parallelism using 128 entries 512-bit vector registers divided among the threads, thus 32 entries per thread. The VPU can do one load and one operation in the same cycle. Each VPU consists of 8 master ALUS, each containing two SP and one DP ALU with independent pipelines: DP pipeline SP pipeline

SAMPLE PAGE FOR PREVIEW

Each vector can contain sixteen 32-bit SP integers or floats, or eight 64-bit DP double integer or floats,

Each vector register is 512 bits wide. For explanation, it is simple to consider that each vector is divided into four lanes with each lane being 128 bits (consisting of four 32 bit elements read from right to left), thus LANE 0 has four 32 bit elements a, b, c, d, and so on.

```
   128 bits           128 bits        128 bits       128 bits
 32b 32b 32b 32b
| D   C   B   A | D   C   B   A | D   C   B   A |  D   C   B   A|
+---------------+---------------+---------------+---------------+
|    LANE 3     |    LANE 2     |    LANE 1     |    LANE 0     |
+---------------+---------------+---------------+---------------+
|  p  o  n  m   |   l  k  j  i  |  h  g  f  e   |  d  c  b  a   |
+---------------+---------------+---------------+---------------+
```

Consider the operation **vmovdqa32 zmm1,zmm0{cdab}** where {cdab} is called the swizzle operation to be performed on the input vector register zmm0 before being output to vector register zmm1. Each vector register can be viewed as having four lanes. Each lane has four 32 bit elements identified by the letters A through D, they are read from right to left, as the right-most has the lowest address. In the 512 bit vector, there are 16 elements denoted by letters a through p.

The vectors are stored in the memory such that the lowest address is on the right-most side. For example, when loading a 32-bit full vector from memory address 0xC000, the first element location 0xC03C.

In the instruction syntax, a swizzle is an optional argument; when specified, the operation performs data element rearrangement or permutations of the source operands before operating on the data element. The swizzle modifies the source operand by creating a copy of the input and generating a multiplexed data pattern using the temporary value and feeding it to the ALU as a source for the operation. The original source operand is not modified and the temporary value is discarded after completing the operation.

For register source swizzle, the supported swizzle operations are described in Table 3.3.1, where {dcba} denotes the four 32-bit elements that form one 128-bit block in the source, reading it right to left, with a being least significant and d being most significant. replicated to all four elements of the same lane. When the source is a register, this functionality is the same for both integer and floating-point instructions. The first few swap patterns in the table ({cdab}, {badc}, {dacb}) are used to shuffle elements within a lane for arithmetic manipulations such as cross-product, horizontal

SAMPLE BOOK PREVIEW

3-4 MIC Architecture Superscalar and Pipelines

In the previous subchapter "MIC Cores Threads and Cache Organization", we have seen the MIC architecture from an administrative point of view: the cores and their logical processors as laid out in a ring topology to interact with shared memory. We also showed how to execute intrinsic instructions on the 512-bit vectors of the VPU.

In this section, we will look at the vector processing unit and its pipelines, you will gain basic understanding on the importance of using homogeneous data type in your code to avoid extra clock cycles penalty.

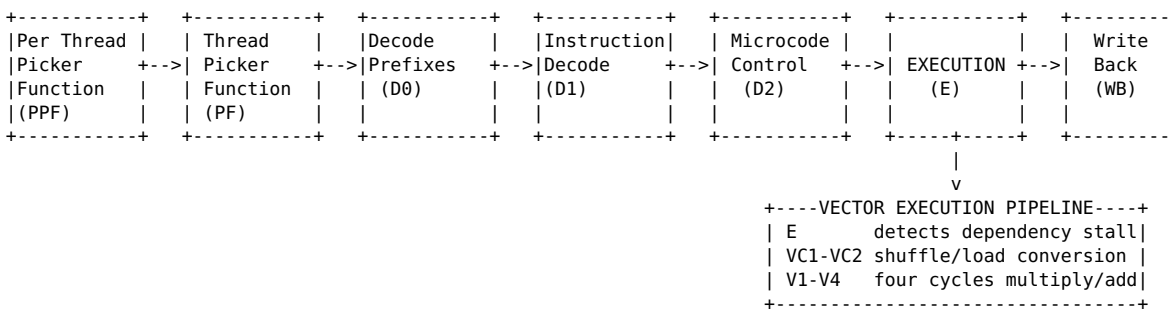
■ MIC's cores VPUs and the Vector Execution Pipelines

Each VPU instruction passes through one or more of the following five pipelines to completion:
- Double-precision (DP) pipeline executes float64 arithmetic, conversion from float64 to float32, and DP-compare instructions.
- Single-precision (SP) pipeline Executes most of the instructions including 64-bit integer loads, float32/int32 arithmetic and logical operations, shuffle/broadcast, loadunpack, type conversions from float32/int32 pipelines, extended math unit (EMU) transcendental instructions, int64 loads, int64/float64 logical, and other instructions.
- Mask pipeline executes mask instructions with 1-cycle latency.
- Store pipeline executes the vector store operations.
- Scatter/gather pipeline executes the vector register read/writes from sparse memory locations.

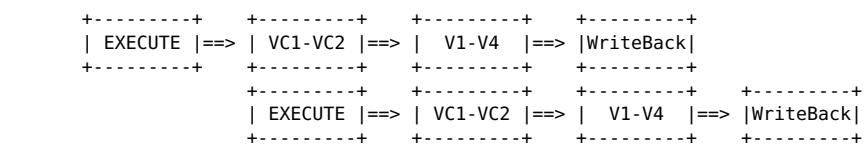
SAMPLE BOOK PREVIEW

Because there are no bypasses between the DP pipeline and the SP pipeline, it is recommended to keep your data type consistent to take advantage of the MIC vector processing. Say you have an instruction in the DP pipeline, followed by a dependent instruction executing in an SP pipeline, the DP instruction has to complete the write before the dependent SP instruction can execute. The SP instruction is stalled until the DP instruction completes its write back, incurring the full 7-cycle delay.

The vector pipeline stages relative to the core pipeline:

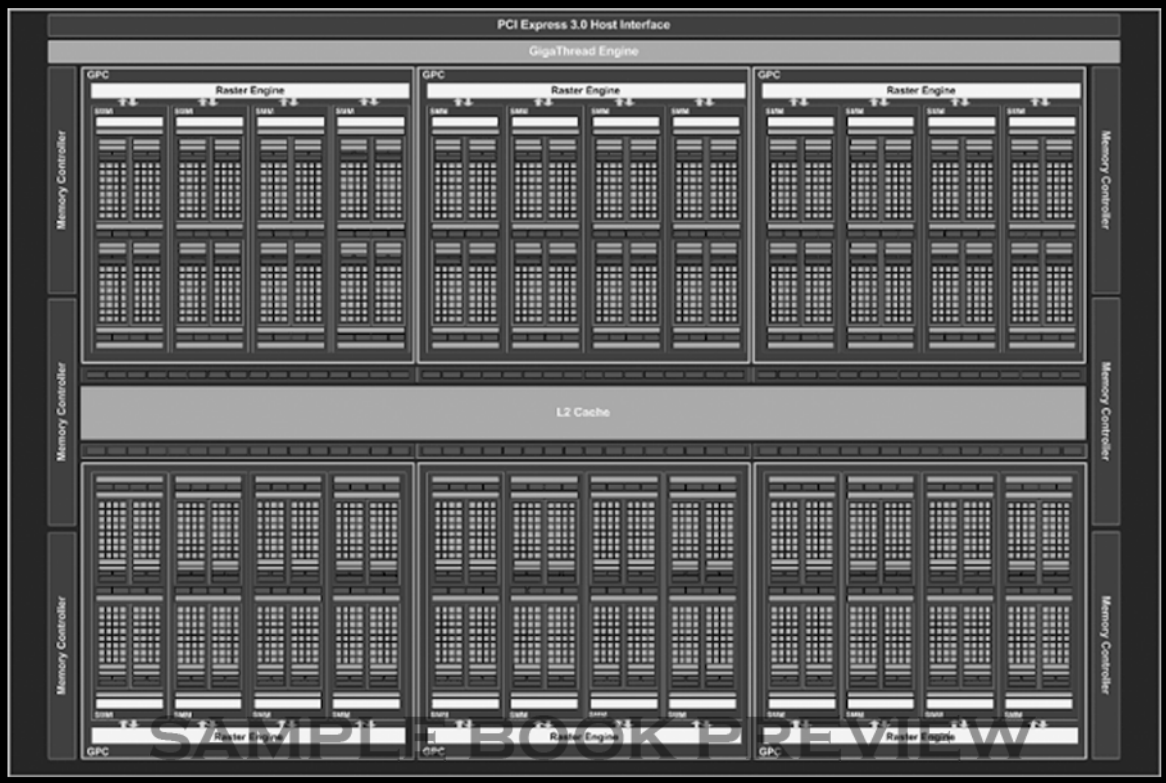


Shown below the execution of two independent SP/SP instructions, where the pipeline can throughput 1 instruction per cycle, with each instruction execution incurring a latency of 4 or more cycles. The 4-cycle minimum constitutes a vector operation on registers spent only in the MADD computation as V1-V4. Any additional shuffle or writing back to memory will add more cycles to the four cycles.



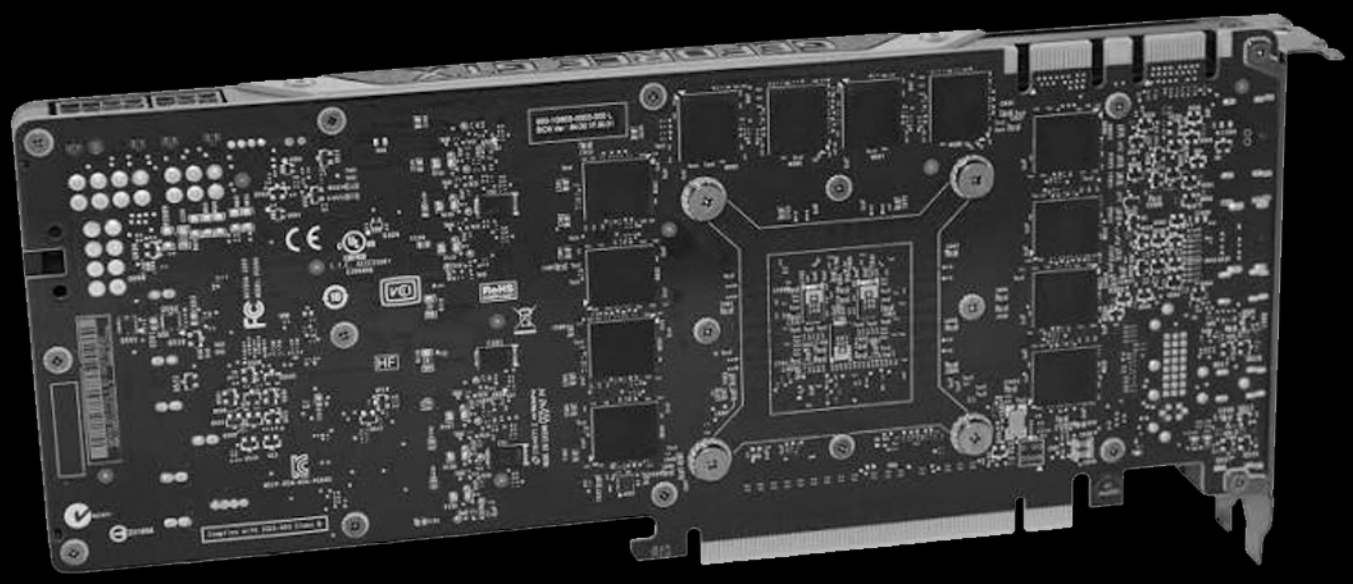
If there are data dependencies for two consecutive SP instructions, the second instruction will wait until data are produced at stage V4, where they will be passed over to the V1 stage of the second instruction using an internal bypass. For an SP instruction followed by a dependent SP instruction,

SAMPLE BOOK PREVIEW



SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW



Chapter 4
CUDA Programming over the HPMC GPU

The HPMC is equipped with a GPU that allows you to do CUDA programming. CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model that makes using a GPU for general purpose computing simple.

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

4-1 System Configuration for the NVIDIA GPU on Your HPMC

Install the NVIDIA GPU on your HPMC. Once slotted in the PCIe, you should be able to see it in the /proc/driver/nvidia.

08:32 root@HPMC: ~ # **tree /proc/driver/nvidia**

```
08:32 root@HPMC: ~ #  tree /proc/driver/nvidia
/proc/driver/nvidia
├── gpus
│   └── 0000:65:00.0
│       ├── information
│       └── registry
├── params
├── patches
│   └── README
├── registry
├── version
└── warnings
    └── README

4 directories, 7 files
```

The GPU can also be seen in the list of PCI.

08:34 root@HPMC: ~ # **lspci -s 65:00.0 -v**

```
08:34 root@HPMC: ~ #  lspci -s 65:00.0 -v
65:00.0 VGA compatible controller: NVIDIA Corporation GM200 [GeForce GTX TITAN X] (rev a1) (prog-if 00 [VGA controller
        Subsystem: NVIDIA Corporation Device 1132
        Flags: bus master, fast devsel, latency 0, IRQ 51, NUMA node 0
        Memory at d7000000 (32-bit, non-prefetchable) [size=16M]
        Memory at c0000000 (64-bit, prefetchable) [size=256M]
        Memory at d0000000 (64-bit, prefetchable) [size=32M]
        I/O ports at b000 [size=128]
        [virtual] Expansion ROM at d8000000 [disabled] [size=512K]
        Capabilities: [60] Power Management version 3
        Capabilities: [68] MSI: Enable+ Count=1/1 Maskable- 64bit+
        Capabilities: [78] Express Legacy Endpoint, MSI 00
        Capabilities: [100] Virtual Channel
        Capabilities: [250] Latency Tolerance Reporting
        Capabilities: [258] L1 PM Substates
        Capabilities: [128] Power Budgeting <?>
        Capabilities: [420] Advanced Error Reporting
        Capabilities: [600] Vendor Specific Information: ID=0001 Rev=1 Len=024 <?>
        Capabilities: [900] #19
        Kernel driver in use: nvidia
        Kernel modules: nouveau, nvidia_drm, nvidia
```

SAMPLE BOOK PREVIEW

To find its memory mapping, we will issue the following commands.

08:41 root@HPMC: ~ # **cat /proc/iomem**
grep '0000:65:00.0'

```
08:41 root@HPMC: ~ #  cat /proc/iomem | grep '0000:65:00.0'
    c0000000-cfffffff : 0000:65:00.0
    d0000000-d1ffffff : 0000:65:00.0
    d7000000-d7ffffff : 0000:65:00.0
    d8000000-d807ffff : 0000:65:00.0
```

08:41 root@HPMC: ~ # **cat /proc/vmallocinfo**
grep phys=d0000000

```
08:41 root@HPMC: ~ #   cat /proc/vmallocinfo | grep phys=d0000000
0xffffc9000c000000-0xffffc9000d001000 16781312 os_map_kernel_space.part.6+0x105/0x130 [nvidia] phys=d0000000 ioremap
08:41 root@HPMC: ~ #   cat /proc/vmallocinfo | grep phys=d7000000
0xffffc9000a000000-0xffffc9000b001000 16781312 os_map_kernel_space.part.6+0xf5/0x130 [nvidia] phys=d7000000 ioremap
```

To print information about the GPU.

08:42 root@HPMC: ~ # **cat /proc/driver/nvidia/gpus/*/information**

```
08:42 root@HPMC: ~ #  cat /proc/driver/nvidia/gpus/*/information
Model:           GeForce GTX TITAN X
IRQ:             51
GPU UUID:        GPU-41a35506-3d8a-4c76-9f36-55cfff039724
Video BIOS:      84.00.1f.00.01
Bus Type:        PCIe
DMA Size:        40 bits
```

SAMPLE BOOK PREVIEW

4-2 CUDA Programming for Your HPMC GPU

The HPMC computer has a GPU that allows you to run CUDA related programs. Your HPMC has been already configured with the essential device drivers so that the GPU can serve you to run any of the following: torch, pytorch, theano, caffe, cuda, and pycuda.

■ Sample CUDA Program

Consider the following program.

-LIS- Listing. 4.2.1 [Sample CUDA Program sample.cu]

```
1.     #include <stdio.h>
2.     #include <cuda.h>
3.
4.     // kernel executes on the CUDA device
5.     __global__ void square_array(double *a, int N) {
6.         int idx = blockIdx.x * blockDim.x + threadIdx.x;
7.         if (idx<N) a[idx] = a[idx] * a[idx];
8.     }
9.
10.    // main routine executes on the host
11.    int main(void) {
12.        double *ptr, *ptr_d;  // pointers to host and device arrays
13.        const int N = 20;  // number of elements in arrays
14.        size_t size = N * sizeof(double);
15.        // allocate array on host
16.        ptr = (double *)malloc(size);  // allocate array on host
17.        // allocate array on device
18.        cudaMalloc((void **) &ptr_d, size);
19.        // initialize host array
20.        for (int i=0; i<N; i++) ptr[i] = (double)i;
21.        // copy host array to CUDA device
22.        cudaMemcpy(ptr_d, ptr, size, cudaMemcpyHostToDevice);
23.        // do calculation on device
24.        int block_size = 4;
25.        int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
26.        square_array <<< n_blocks, block_size >>> (ptr_d, N);
27.        // copy array from device to host array
28.        cudaMemcpy(ptr, ptr_d, sizeof(double)*N, cudaMemcpyDeviceToHost);
29.        // print results
30.        for (int i=0; i<N; i++) printf("%d %f\n", i, ptr[i]);
31.        free(ptr); cudaFree(ptr_d);
32.    }
33.
```

SAMPLE BOOK PREVIEW

HPMC 2026

The program is compiled with nvcc with the options -g -G to include symbols for debugging using cuda-gdb.

08:01 root@HPMC: /AxBcuda # **nvcc -g -G -o sample sample.cu**

08:03 root@HPMC: /AxBcuda # **./sample**

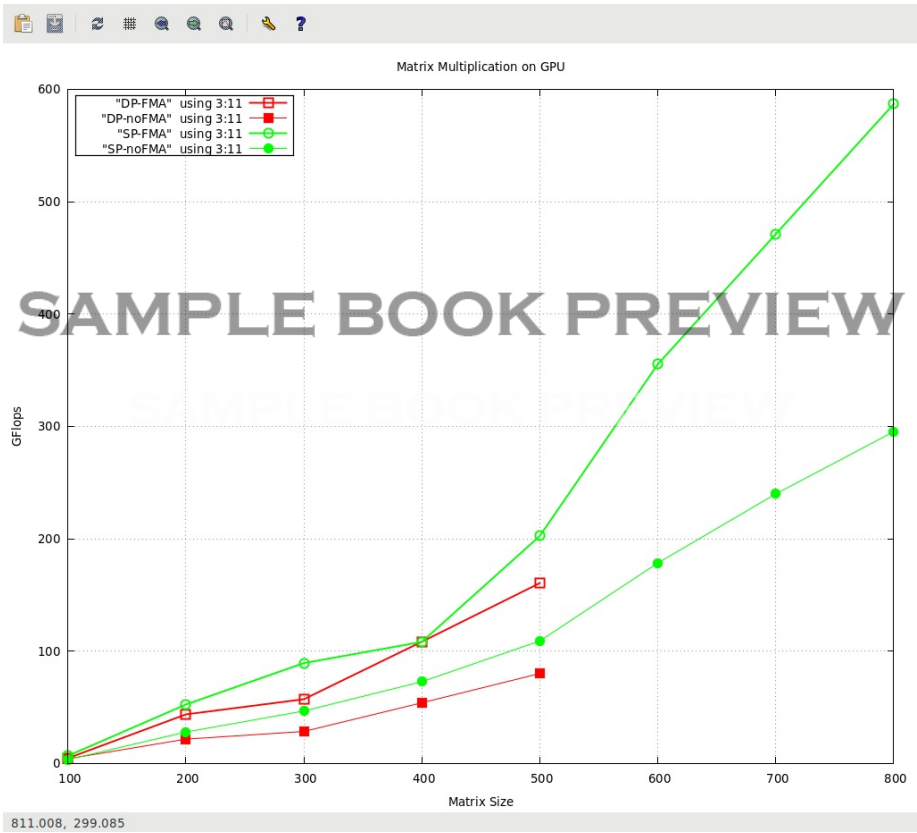
```
0 0.000000
```

SAMPLE BOOK PREVIEW

4-3 Measuring Gigaflops for Matrix Multiplication on the GPU

Figure 4.1.3 shows the result of matrix multiplication on the GPU. The plots are for the operations performed on various matrix sizes, considering operations to be performed as FMA and noFMA on either double precision or single precision.

The result is explicitly drawn from the programming exercise explained in this section.



SAMPLE BOOK PREVIEW

Fig. 4.3.1 [Matrix Multiplication Benchmarked on GPU]

HPMC 2026 by Bassem Jamaleddine

■ Measuring Gigaflops for Matrix Multiplication with Double Precision on the GPU

Consider the following program.

-LIS- Listing. 4.3.1 [Matrix Multiplication on GPU with Double Precision: AxBd.cu]

```
#include<cuda.h>
#include<stdio.h>
#include <sys/time.h>
```

SAMPLE BOOK PREVIEW

4-4 CUDA Fortran

FORTRAN programs can link to external CUDA functions. In this chapter we shows two FORTRAN programs: the first VECADDfort.f95 that links to CUDA external function to add two vectors, and the other AxBfort.f95 that links to CUDA external function to multiply two matrices.

■ FORTRAN Program Calling CUDA to Add Two Vectors

FORTRAN program can call CUDA routines by linking to CUDA object files. The following shows an example on how to compile a FORTRAN program that calls an external CUDA function through the cuda_kernel_wrapper method.

-LIS- Listing. 4.4.1 [FORTRAN Program to Add Vector Calling CUDA Function]

```
1.     PROGRAM VECADDfort
2.
3.     ! simple program which creates 2 vectors and adds them in an
4.     ! external cuda function
5.
6.     IMPLICIT NONE
7.
8.     integer*4 :: I
9.     integer*4, parameter :: N=8
10.    real*4, Dimension(N) :: A, B
11.
12.    DO I = 1, N
13.        A(I) = I * 1.0
14.        B(I) = 1.5
15.    END DO
16.
17.    print *, 'A       = ', (A(I), I=1,N)
18.
19.    CALL cuda_kernel_wrapper(A, B, N)
20.
21.    print *, 'A + 1.5 = ', (A(I), I=1,N)
22.
23.    END PROGRAM
```

SAMPLE BOOK PREVIEW

HPMC 2026

CUDA program with an external function callable from FORTRAN programs.

-LIS- Listing. 4.4.2 [CUDA Program With External Function to Add Two Vectors]

```
1.     #include <stdio.h>
2.     #include <stdlib.h>
3.     #include <string.h>
4.     #include <cuda.h>
5.     #include <cuda_runtime.h>
6.
7.     // kernel function to add two vectors
8.     __global__ void vect_add(float *a, float *b, int N) {
9.         int idx = threadIdx.x;
10.        if (idx<N) a[idx] = a[idx] + b[idx];
11.    }
12.
13.    // function called from main's fortran program
14.    extern "C" void cuda_kernel_wrapper_(float *a, float *b, int *Np) {
15.        float  *a_d, *b_d;  // device GPU vectors
16.
17.        int blocks = 1;        // uses 1 block of
18.        int N = *Np;        // N threads on GPU
19.
20.        // allocate memory on GPU
21.        cudaMalloc( (void **)&a_d, sizeof(float) * N );
22.        cudaMalloc( (void **)&b_d, sizeof(float) * N );
23.
24.        // copy vectors from CPU to GPU
25.        cudaMemcpy( a_d, a, sizeof(float) * N, cudaMemcpyHostToDevice );
26.        cudaMemcpy( b_d, b, sizeof(float) * N, cudaMemcpyHostToDevice );
27.
```

SAMPLE BOOK PREVIEW

4-5 CUDA Application for Your HPMC GPU

The HPMC computer has a GPU that allows you to code and run CUDA related programs. Your HPMC has been already configured with the essential device drivers so that the GPU can serve you to run any of the following application programming languages: torch, pytorch, theano, caffe, cuda, and pycuda.

■ CUDNN with Torch

You program in Lua and run torch programs on your HPMC.

torch has a tensor library like NumPy, with strong GPU support. PyTorch is used either as a replacement for NumPy to use the GPU as a processing agent, or as a platform for deep learning research. The interpreter is run with the two letters command th that is followed by the Lua program.

```
20:09 root@HPMC7: /appz/torch #  th bench.lua

20:09 root@HPMC7: /appz/torch #  th bench.lua
Found Environment variable CUDNN_PATH = /usr/local/cuda-9.2/lib64/libcudnn.so.7.1.4
Running on device: GeForce GTX TITAN X

CONFIG: input = 3x128x128 * ker = 3x96x11x11 (bs = 128, stride = 1)
cudnn.SpatialConvolution               :updateOutput():      51.10
cudnn.SpatialConvolution             :updateGradInput():     57.34
cudnn.SpatialConvolution           :accGradParameters():     50.86
cudnn.SpatialConvolution                       :TOTAL:     159.30
nn.SpatialConvolutionMM                :updateOutput():      44.83
nn.SpatialConvolutionMM              :updateGradInput():     54.59
nn.SpatialConvolutionMM            :accGradParameters():     56.69
nn.SpatialConvolutionMM                        :TOTAL:     156.12
ccn2.SpatialConvolution                :updateOutput():      32.38
ccn2.SpatialConvolution              :updateGradInput():     50.96
ccn2.SpatialConvolution            :accGradParameters():     44.72
ccn2.SpatialConvolution                        :TOTAL:     128.06

CONFIG: input = 64x64x64 * ker = 64x128x9x9 (bs = 128, stride = 1)
cudnn.SpatialConvolution               :updateOutput():      89.60
cudnn.SpatialConvolution             :updateGradInput():     264.64
cudnn.SpatialConvolution           :accGradParameters():     141.98
cudnn.SpatialConvolution                       :TOTAL:     496.21
nn.SpatialConvolutionMM                :updateOutput():     138.05
nn.SpatialConvolutionMM              :updateGradInput():     219.20
nn.SpatialConvolutionMM            :accGradParameters():     330.22
nn.SpatialConvolutionMM                        :TOTAL:     687.47
ccn2.SpatialConvolution                :updateOutput():     109.02
ccn2.SpatialConvolution              :updateGradInput():     125.64
ccn2.SpatialConvolution            :accGradParameters():     227.74
ccn2.SpatialConvolution                        :TOTAL:     462.40

CONFIG: input = 128x32x32 * ker = 128x128x9x9 (bs = 128, stride = 1)
cudnn.SpatialConvolution               :updateOutput():      30.75
cudnn.SpatialConvolution             :updateGradInput():     74.42
cudnn.SpatialConvolution           :accGradParameters():     54.36
cudnn.SpatialConvolution                       :TOTAL:     159.52
nn.SpatialConvolutionMM                :updateOutput():      53.73
nn.SpatialConvolutionMM              :updateGradInput():     103.11
nn.SpatialConvolutionMM            :accGradParameters():     55.03
nn.SpatialConvolutionMM                        :TOTAL:     211.87
ccn2.SpatialConvolution                :updateOutput():      37.26
ccn2.SpatialConvolution              :updateGradInput():     46.71
ccn2.SpatialConvolution            :accGradParameters():     85.09
ccn2.SpatialConvolution                        :TOTAL:     169.07

CONFIG: input = 128x16x16 * ker = 128x128x7x7 (bs = 128, stride = 1)
cudnn.SpatialConvolution               :updateOutput():       4.33
```

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

4-6 GPU Programming with CAFFE

HPMC calculators have **CAFFE = Convolutional Architecture for Fast Feature Embedding** already installed on them.
Caffe is a deep learning framework developed at University of California, Berkeley. It is written in C++, with a Python interface. It is open source, under a BSD license.

Caffe supports many different types of deep learning architectures geared towards image classification and image segmentation. It supports CNN, RCNN, LSTM and fully-connected neural network designs. Caffe supports GPU- and CPU-based acceleration computational kernel libraries such as Nvidia cuDNN and Intel Xeon Phi MIC coprocessor.

Installing CAFFFE is straight forward: download the distribution into a directory, and build it from there:

export LD_LIBRARY_PATH=:/usr/lib/nvidia:/opt/lib64:/opt/lib:/usr/local/cuda/lib64:/usr/lib64/atlas

make all

ls -l .build_release/lib/libcaffe.so

make test

SAMPLE BOOK PREVIEW

make runtest

SAMPLE BOOK PREVIEW

The following shows the testing of thousands of functions of CAFFE installed on the HPMC.

```
.build_release/tools/caffe
caffe: command line brew
usage: caffe <command> <args>

commands:
  train           train or finetune a model
  test            score a model
  device_query    show GPU diagnostic information
  time            benchmark model execution time

  Flags from tools/caffe.cpp:
    -gpu (Run in GPU mode on given device ID.) type: int32 default: -1
    -iterations (The number of iterations to run.) type: int32 default: 50
    -model (The model definition protocol buffer text file..) type: string
      default: ""
    -snapshot (Optional; the snapshot solver state to resume training.)
      type: string default: ""
    -solver (The solver definition protocol buffer text file.) type: string
      default: ""
    -weights (Optional; the pretrained weights to initialize finetuning. Cannot
      be set simultaneously with snapshot.) type: string default: ""
.build_release/test/test_all.testbin 0 --gtest_shuffle
Cuda number of devices: 1
Setting to use device 0
Current device id: 0
Note: Randomizing tests' orders with a seed of 43660 .
[==========] Running 1080 tests from 190 test cases.
[----------] Global test environment set-up.
[----------] 3 tests from ThresholdLayerTest/2, where TypeParam = caffe::FloatGPU
[ RUN      ] ThresholdLayerTest/2.Test2
[       OK ] ThresholdLayerTest/2.Test2 (233 ms)
[ RUN      ] ThresholdLayerTest/2.Test
[       OK ] ThresholdLayerTest/2.Test (0 ms)
[ RUN      ] ThresholdLayerTest/2.TestSetup
[       OK ] ThresholdLayerTest/2.TestSetup (0 ms)
```

SAMPLE BOOK PREVIEW

4-7 GPU Programming with Theano

HPMC calculators have Theano already installed on them. You can program in python and run theano on your HPMC.

Theano is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. Theano combines the convenience of NumPy's syntax.

Make sure the following stanza is in your ~/.theanorc as this will define the option gpu-architecture as sm_52 for the nvcc program.

```
# cat ~/.theanorc

[nvcc]
fastmath = True
flags = -arch=sm_30
```

To have theano started with (base) prompt, you can add the following script to /tools.

SAMPLE BOOK PREVIEW

-LIS- Listing. 4.7.1 [Script to source theano at the command prompt]

SAMPLE Book PREVIEW

```
1.     export THEANO_FLAGS='device=cuda,floatX=float32'
2.     eval "$(/cuda9/miniconda3/bin/conda shell.bash hook)"
```

HPMC 2026

Get the prompt by simpy sourcing the script at the command prompt.

```
20:18 root@HPMC: ~ # source /tools/theano

20:18 root@HPMC: ~ #  source /tools/theano
(base) 20:18 root@HPMC: ~ #  python exthea3.py
```

■ Theano Example

The following show an example of executing a theano script. Consider the following first example of a theano script.

-LIS- Listing. 4.7.2 [Example of a Theano Script]

```
1.     from theano import function, config, shared, sandbox
2.     import theano.tensor as T
3.     import numpy
4.     import time
5.
6.     vlen = 10 * 30 * 768  # 10 x #cores x # threads per core
7.     iters = 1000
8.
```

SAMPLE BOOK PREVIEW

4-8 QUDA - Accelerated lattice Quantum Chromodynamics (LQCD) Calculations on the GPU

The open source library **QUDA = Quantum Chromodynamics on NVIDIA GPUs** has been installed on your HPMC. QUDA is designed to accelerate lattice Quantum Chromodynamics (LQCD) calculations using GPUs via NVIDIA's CUDA platform. It includes solvers for conjugate gradient (CG), multi-shift CG, BiCGStab, BiCGStab(l), and domain-decomposition (DD) preconditioned GCR. Installing QUDA on your HPMC is simple: download quda-master tar archive, extract it, run configure, and make.

From the README file of the package.

```
Release Notes for QUDA v0.7.0                          4th February 2015
----------------------------

Overview:

QUDA is a library for performing calculations in lattice QCD on
graphics processing units (GPUs), leveraging NVIDIA's CUDA platform.
The current release includes optimized Dirac operators and solvers for
the following fermion actions:

* Wilson
* Clover-improved Wilson
* Twisted mass (including non-degenerate pairs)
* Twisted mass with a clover term
* Staggered fermions
* Improved staggered (asqtad or HISQ)
* Domain wall (4-d or 5-d preconditioned)
* Mobius fermion

Implementations of CG, multi-shift CG, BiCGstab, and DD-preconditioned
GCR are provided, including robust mixed-precision variants supporting
combinations of double, single, and half (16-bit "block floating
point") precision.  The library also includes auxilliary routines
necessary for Hybrid Monte Carlo, such as HISQ link fattening, force
terms and clover-field construction.  Use of many GPUs in parallel is
supported throughout, with communication handled by QMP or MPI.
```

SAMPLE BOOK PREVIEW

The quda-master tree is shown below:

09:30 root@skyblue: /hpmc/cuda/quda-master # **tree --charset unicode -d**

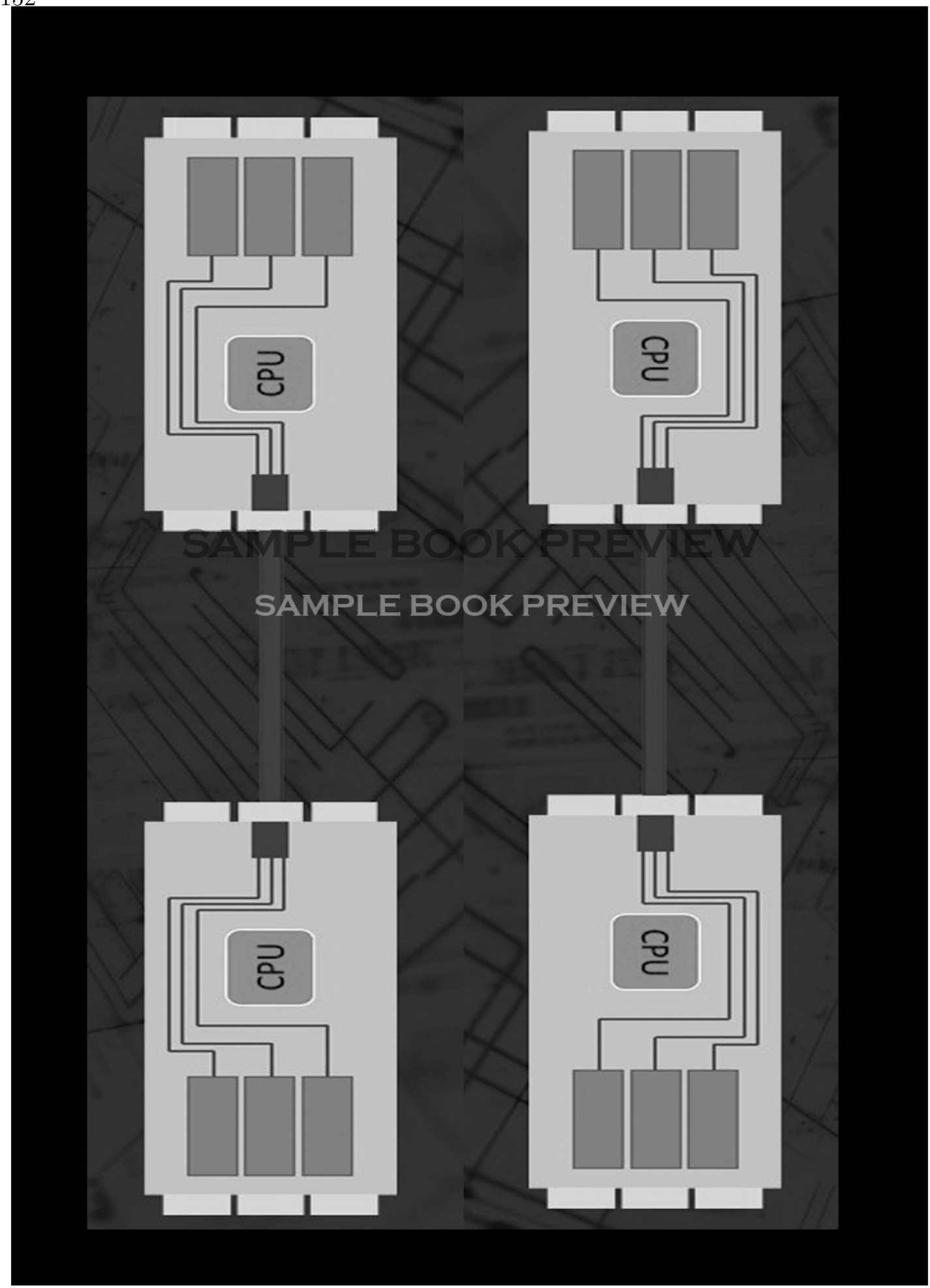
```
.
|-- include
|   `-- cub
|       |-- block
|       |   `-- specializations
|       |-- device
|       |   |-- block
|       |   |   `-- specializations
|       |   `-- region
|       |       `-- specializations
|       |-- grid
|       |-- host
|       |-- iterator
|       |-- thread
|       `-- warp
|           `-- specializations
|-- lib
|   |-- dslash_core
|   `-- generate
`-- tests
```

To find out what the quda-master package offers you, just look into its library directory for the *.cu files. The filenames in the lib directory give you a good idea on what is provided by quda-master.

10:25 root@skyblue: /hpmc/cuda/quda-master # **ls -l lib/*.cu | lsnames**

```
blas_quda
clover_deriv_quda
clover_invert
clover_quda
clover_trace_quda
color_spinor_util
contract
copy_clover
copy_color_spinor
copy_gauge
copy_gauge_double
copy_gauge_extended
```

SAMPLE BOOK PREVIEW



SAMPLE BOOK PREVIEW
SAMPLE BOOK PREVIEW

Chapter 5
Configuring Infiniband for RDMA

The HPMC can be extended to access neighboring nodes memory via Infiniband appliances for RDMA access. This is realized through a switch connectivity that requires the following components:
The switch:
Qlogic 12200-18 18-Port QDR InfiniBand Network Switch 12200-18-28

The cables:
Intel Qlogic CBL2-1001001-3, 40G QSFP+ Active Optical Cable QDR
Qlogic CBL1-0600230 2M Cable Cop 30 AWG STD QSFP to QSFP
EMC 003-0080-01 3 Meter Infiniband 40gb QSFP to Cx4
EMC 003-0080-01 3 METER 40gb QSFP to Cx4 Cable

The Host Channel Adapters:
QLogic QLE7340 1-Port 40Gb QSFP Infiniband QDR PCIe HCA Host Channel Adapter
QLogic InfiniPath QLE7140 IB6110401-01 InfiniBand SDR 4x 10Gb PCI-e x8

SAMPLE BOOK PREVIEW
SAMPLE BOOK PREVIEW

5-1 Configuring Infiniband for RDMA

The QLogic switch, shown in Figure 5.1.1, is combined with HCA communication adapters to connect HPMCs as compute nodes in high-performance parallel architectures. That is, when the HPMC is equipped with an HCA, it can be interconnected to other HPMCs over the switch to participate as a computing node to run programs in parallel.
Previously, we used OpenMP to run programs in parallel withinone single HPMC node, sharing memory while multi-threading via the HPMC's CPU cores or its MIC's cores. OpenMP is limited to one single node, whether we run the program locally on the HPMC or on its MIC, sharing the memory on either device.
To run programs in parallel on the HPMC and on its MIC in parallel, we need to use MPI (Message Passing Interface). To run programs in parallel on multiple HPMC nodes connected by a network, we need to use MPI.

To add the InfiniBand verbs to the HPMC, you need to install and configure the folowing packages.
The OFED packages required are:

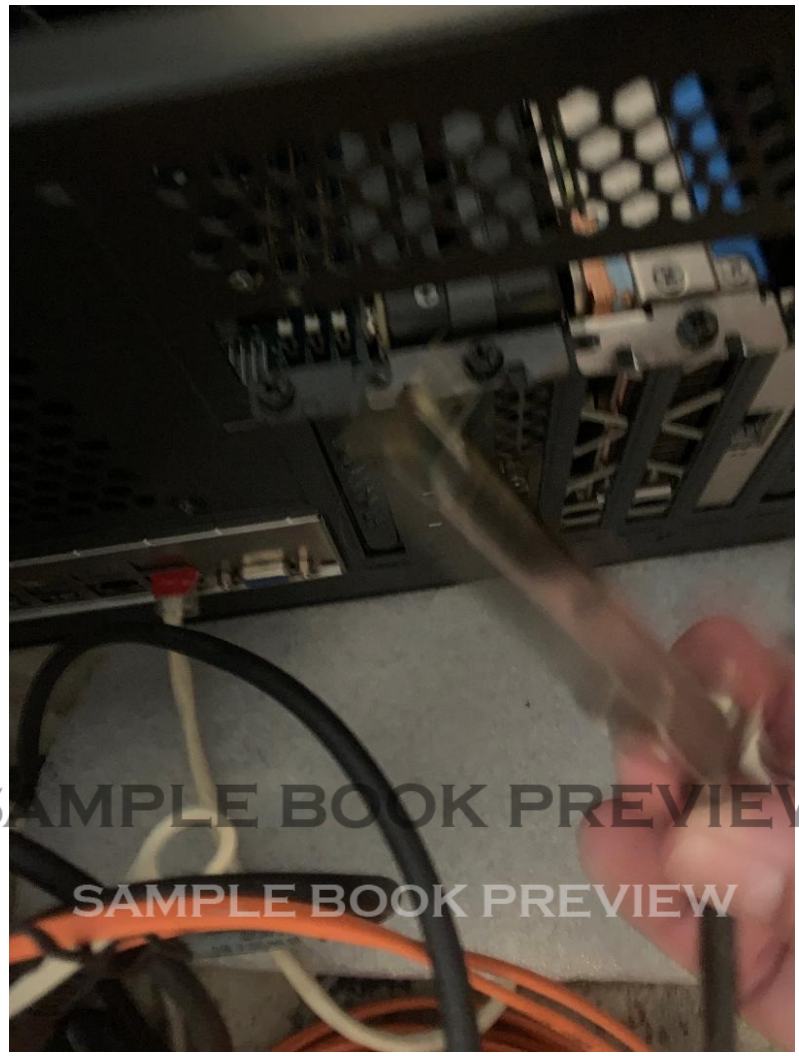
```
ofed-scripts
rdma-core
rdma-core-devel
libibverbs
libibverbs-utils
librdmacm
librdmacm-utils
libibumad
ibacm
iwpmd
srp_daemon
mstflint
ofed-docs
compat-rdma
```

SAMPLE BOOK PREVIEW
SAMPLE BOOK PREVIEW



Fig. 5.1.1 [The switch]

HPMC 2026 by Bassem Jamaleddine



SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

Fig. 5.1.2 [Connecting hpmc7 to the switch via its HCA]

HPMC 2026 by Bassem Jamaleddine

■ Probe The Infiniband HCA Cards

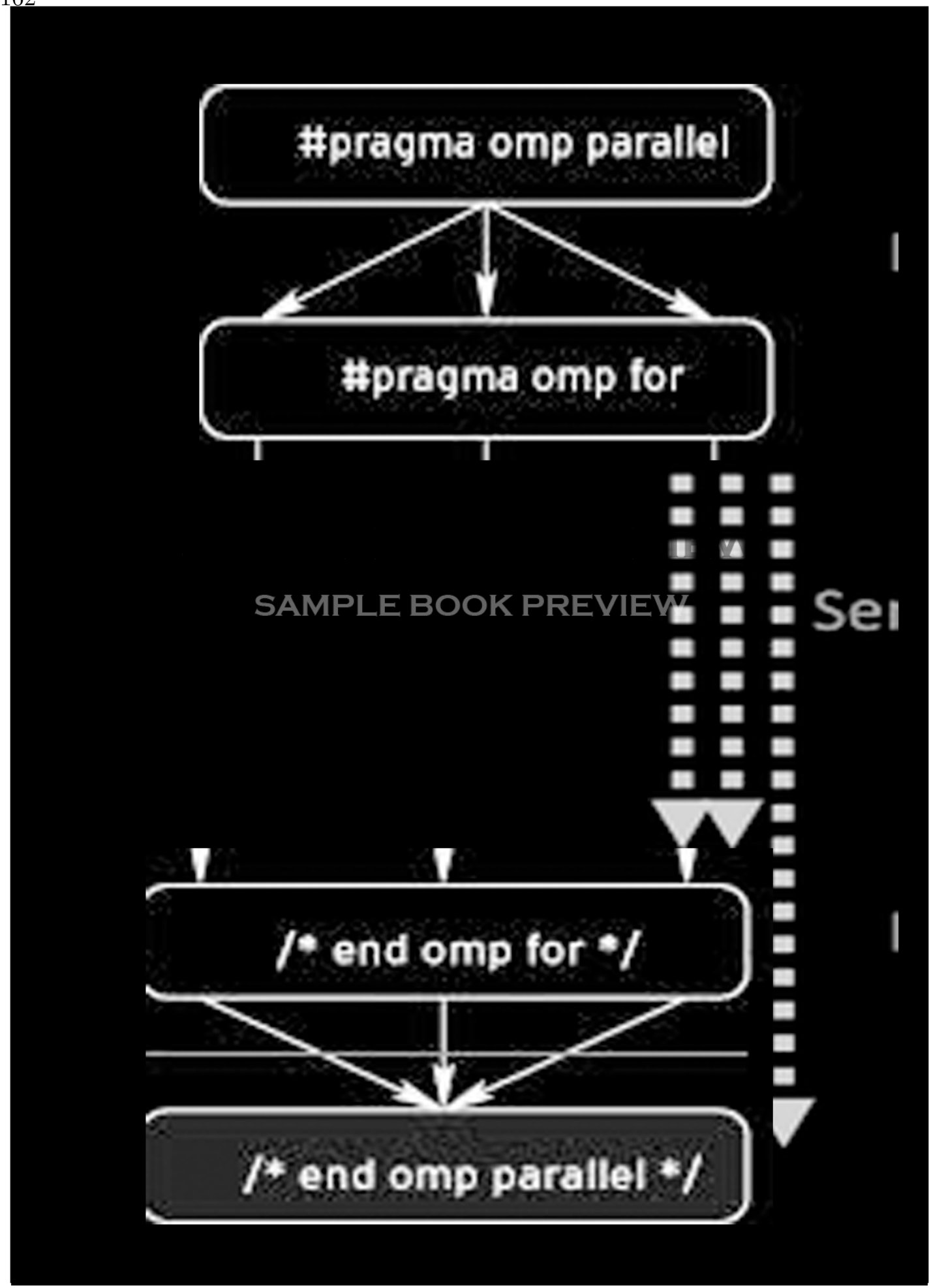
```
# lspci -nn | grep HCA
```

```
03:03 root@HPMC9: /mm03fs/MPI #  lspci -nn %%%pipe grep HCA
5e:00.0 InfiniBand [0c06]: QLogic Corp. IBA7322 QDR InfiniBand HCA [1077:7322] (rev 02)
5f:00.0 InfiniBand [0c06]: QLogic Corp. IBA7322 QDR InfiniBand HCA [1077:7322] (rev 02)
```

```
#  grep -i 1fc1 /lib/modules/3.10.0-693.el7.x86_64/modules.*
```

```
03:07 root@HPMC9: /mm03fs/MPI #  grep -i 1fc1 /lib/modules/3.10.0-693.el7.x86_64/modules.*
/lib/modules/3.10.0-693.el7.x86_64/modules.alias:alias pci:v00001FC1d0000000Dsv*sd*bc*sc*i* ib_ipath
/lib/modules/3.10.0-693.el7.x86_64/modules.alias:alias pci:v00001FC1d00000010sv*sd*bc*sc*i* ib_qib
```

SAMPLE BOOK PREVIEW



SAMPLE BOOK PREVIEW

Chapter 6
RDMA Over Infiniband Connectivity

Remote Direct Memory Access (RDMA) is a high-performance networking technology that allows a device to transfer data directly to or from the memory of another device, bypassing both hosts' operating systems, CPUs, and caches. This results in ultra-low latency, high throughput, and lower CPU overhead, making it essential for AI/ML training, HPC, and fast storage, often using InfiniBand or RoCE.

RoCE (RDMA over Converged Ethernet) and InfiniBand (IB) are both high-performance, low-latency networking technologies utilizing RDMA to bypass CPU kernels, primarily used in AI and HPC clusters. InfiniBand offers superior performance and a native lossless, simplified architecture, while RoCE provides cost-effective scalability, leveraging existing Ethernet infrastructure.

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

6-1 RDMA Over Infiniband Connectivity

HPMC calculators that are fabric capable can be interconnected together to take advange of RDMA via DAPL to execute programs and distribute the processing on neighboring HPMC's.

When the HPMC is equipped with a switching card device named qib0, it is capable to communicate with a neighboring HPMC that is equipped with a similar card via the switch.

To find out if your HPMC is RDMA capable and it is running properly, you need to verify that "IB/RoCE v1" is up. The following steps show how to determine if your HPMC is "IB/RoCE v1" capable.

Make sure opensm is running:

```
# systemctl status opensm
```

Make sure DAPL is running and RoCE is running. To validate that RoCE "IB/RoCE v1" is up:

```
# mkdir /sys/kernel/config/rdma_cm/qib0
```

```
# tree /sys/kernel/config/rdma_cm/qib0
```

```
# cat /sys/kernel/config/rdma_cm/qib0/ports/1/default_roce_mode
```

The above command should print: **"IB/RoCE v1"** as shown below.

```
#  mkdir /sys/kernel/config/rdma_cm/qib0
#  tree /sys/kernel/config/rdma_cm/qib0
/sys/kernel/config/rdma_cm/qib0
└── ports
    └── 1
        └── default_roce_mode

2 directories, 1 file
#  cat /sys/kernel/config/rdma_cm/qib0/ports/1/default_roce_mode
IB/RoCE v1
```

SAMPLE BOOK PREVIEW

The device qib0 is configured on the network as ib0. Any of the following commands can be used to query ib0:

```
# ip link show ib0
```

```
# ifconfig ib0
```

```
# ibv_devinfo -v qib0
```

```
#  ip link show ib0
5: ib0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 2044 qdisc pfifo_fast state UP mode DEFAULT group default qlen 256
    link/infiniband 80:00:00:03:fe:80:00:00:00:00:00:00:00:11:75:00:00:6e:ef:26 brd 00:ff:ff:ff:ff:12:40:1b:ff:ff:00:00:00:
#  ifconfig ib0
ib0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 2044
        inet 172.16.0.132  netmask 255.255.0.0  broadcast 172.16.255.255
        inet6 fe80::211:7500:6e:ef26  prefixlen 64  scopeid 0x20<link>
Infiniband hardware address can be incorrect! Please read BUGS section in ifconfig(8).
        infiniband 80:00:00:03:FE:80:00:00:00:00:00:00:00:00:00:00:00:00:00:00  txqueuelen 256  (InfiniBand)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

#  ibv_devinfo -v qib0
hca_id: qib0
        transport:                      InfiniBand (0)
        fw_ver:                         0.0.0
        node_guid:                      0011:7500:006e:ef26
        sys_image_guid:                 0011:7500:006e:ef26
        vendor_id:                      0x1175
...
```

Make sure that it is set to **datagram** mode (this configuration is set in the network scripts: ifcnf-ib0 ifcnf-ib0.8002

```
# cat /sys/class/net/ib0/mode
```

To change the mode to **connected**, edit the ifcnf-ib0 ifcnf-ib0.8002 and reboot your HPMC.

HPMC calculators branded as HPMC-KLIB, KLIB Knight-Landing InfiniBand, are all fabric connectivity ready and can be plugged to the switch.

The following sections show some of the commands to test and measure the performance of RDMA between the nodes.

■ Testing Connectivity with ping

SAMPLE BOOK PREVIEW

6-2 MPI Programming with Fabric MPIRUN Offloading to MIC

HPMC calculators that are fabric capable can be interconnected together to take advange of RDMA via DAPL (direct access programming library) to execute programs and distribute the processing on neighboring HPMCs.

It is possible to interconnect two or more HPMCs DMA using switched network cards, called infinibands. An application that is executed on the MIC of an HPMC can also be executed on the MIC of another HPMC system. This parallel programming is possible with MPI.

Parallel programming with MPI, Message Passing Interface, can be performed on your HPMC calculator. If you have two or more HPMC calculators interconnected with fabric wiring, then you can scale the execution of your MPI program by allocating more threads.

In MPI, the programmer does not need to focus on shared memory as each processor has its own private memory. Processors communicate between each other by explcitly sending messages (trhough communicators). Each process within a communicator has a unique rank identified by an integer starting at zero. When running the MPI program, one can specify a number for the ranks. The message passing between processes permits the exchange of data, while the sending of messages can contain buffers of data specifying the receiving process rank.

The core (* not CPU core) of such execution is laid through RDMA basic and essential components, all known to be stacked in what is called rdma-core. Message passing across the RDMA is essentially what allows the software application to be executed on a group (* we will avoid using cluster or clustering) of HPMC. The Message Passing Interface (MPI) is a set of RFC protocol (open or not open) that allows the execution of a program to be distributed on many HPMCs. While MPI is a monotonic interface that explores the RDMA and IPoIB switching (subject of DAPL) its execution benefits from leaving the Central Processing Unit (CPU) alone, hence it is asynchronous and fully duplexed depending on the Net-InfiniBand.

SAMPLE BOOK PREVIEW

Consider the following trap_offload.c program that we will be running to test the MPI execution on our HPMCs.

-LIS- Listing. 6.2.1 [Trap Offload Program]

```
1.     #include <math.h>
2.     #include <mpi.h>
3.     #include <stdio.h>
4.     #include <unistd.h>
5.     #define NUM_TRAPEZOIDS 10000000000
6.     __attribute__((target(mic))) inline double f(double x) {
7.         return 1.00*x*x*exp(-(x-0.0)*(x-0.0)/(2.0*0.25*0.25))
8.             + 0.50*x*x*exp(-(x-0.2)*(x-0.2)/(2.0*0.50*0.50))
9.             + 0.50*x*x*exp(-(x+0.2)*(x+0.2)/(2.0*0.50*0.50))
10.            + 0.25*x*x*exp(-(x-0.4)*(x-0.4)/(2.0*1.00*1.00))
11.            + 0.25*x*x*exp(-(x+0.4)*(x+0.4)/(2.0*1.00*1.00));
12.    }
13.
14.    int main (int argc, char *argv[]) {
15.        int namelen, rank, size;
16.        char name[MPI_MAX_PROCESSOR_NAME];
17.        double upper_bound = 5.0, lower_bound = -5.0;
18.        double x0, x1, width;
19.        double integral = 0;
20.        double compute_time, total_time;
21.        int chunk_size;
22.        MPI_Init(&argc, &argv);
23.        MPI_Comm_size(MPI_COMM_WORLD, &size);
24.        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
25.        MPI_Get_processor_name(name, &namelen);
26.        chunk_size = NUM_TRAPEZOIDS / size;
27.        x0 = lower_bound+(upper_bound-lower_bound)*rank/size;
28.        x1 = x0 + (upper_bound - lower_bound)/size;
29.        width = (x1-x0)/chunk_size;
30.        MPI_Barrier(MPI_COMM_WORLD);
31.        compute_time = total_time = MPI_Wtime();
32.    #pragma offload target(mic)
33.    #pragma omp parallel
34.    #pragma omp for reduction(+:integral)
35.        for (int i = 0; i < chunk_size ; i++) {
36.            integral += 0.5*width *(f(x0+width*i)+f(x0+width*(i+1)));
37.        }
38.        compute_time = MPI_Wtime() - compute_time;
39.        MPI_Allreduce(MPI_IN_PLACE, &integral, 1, MPI_DOUBLE, MPI_SUM,
       MPI_COMM_WORLD);
40.        total_time = MPI_Wtime() - total_time;
41.        printf("rank %d of %d on %s: %f seconds\n", rank, size, name,
       compute_time);
42.        if (rank == 0) {
43.            printf("integral = %f, time = %f\n", integral, total_time);
44.        }
45.        MPI_Finalize();
```

SAMPLE BOOK PREVIEW

A Logging in to the MIC

MIC Essentials:

The MIC card is Intel Math Co-Processor that is slotted in PCIe and is made available on your HPMC host computer. The HPMC is the host computer and the MIC card is the client computer. At any time you should be able to ssh to the host MIC simply by typing: ssh mic0

This section introduces you to the essential administration of your MIC Co-processor-.

The MIC card is booted within your own HPMC as an embedded Linux system. There are many steps to get the MIC operational. Understanding these steps will help you to troubleshoot issues that you encounter when using your MIC Co-Processor.

First step, try to ssh to your MIC:

ssh mic0

You should be logged into the MIC and presented with the MIC Linux prompt. Such login does not require to provide any user name or password. If you cannot login to your MIC, then make sure that MPSS is up and running:

systemctl status mpss

SAMPLE BOOK PREVIEW

You can restart MPSS and try to ssh mic0 again:

SAMPLE BOOK PREVIEW

systemctl restart mpss

Check that MPSS is up and serving, you can tail the dmesg in another terminal while "systemctl restart mpss" is being restarted:

tail dmesg

If MPSS fails to restart then reset the MIC card:

micctrl -rw

The above command will reset the MIC and wait until the reset is done. Now check on your MIC status. The MIC status must be in "ready" state before starting MPSS. Check the status of the MIC:

micctrl -s

If the above command shows that the MIC is in "ready" state, then you can retry starting MPSS:

systemctl restart mpss

Wait a minute then check if MPSS service is brought up successfully:

systemctl status mpss

Also you can check the system dmesg output:

dmesg

Now you should be able to ssh to your MIC:

ssh mic0

If you cannot "ssh mic0" successfully, then you need to regenerate your id_rsa. Assuming you are the user 'root', issue the following command to regenerate your id_rsa:

SAMPLE BOOK PREVIEW

B Troubleshooting the MIC

The MIC is booted as a computer within your HPMC computer system. Your HPMC computer is the host computer that will treat the MIC Co-Processor as a computer plugged into its PCI bus, that will embed its OS image, for this to happen:

- the MIC device driver must be loaded, hence to communicate and control the MIC hardware layer. From your HPMC verify the mic module is loaded (this is true for x100 series)
  # lsmod | grep mic
- the MIC is recognized as a device to the host computer. You can simply see if the mic0 device is there:
  # ls /dev/mic0
- the MPSS software layer must be available on the HPMC host computer. A simple way to verify that MPSS is installed, just see if micinfo can be recognized on your command prompt:
  # which micinfo

To bring the MIC up

* it is embedded system whose root directory is mapped to /var/mpss/mic0, hence the root user directory that will be available when MIC Co-Processor is bootstrapped can be seen on your HPMC host: /var/mpss/mic0/root
  # ls -la /var/mpss/mic0/root

Hence basic changes to the MIC root diretory are possible:

- Generate the key on the host and add it to the authorized_keys on the MIC mapped directory:
    ssh-keygen
    cat /root/.ssh/id_rsa.pub > /var/mpss/mic0/root/.ssh/authorized_keys
- micctrl --resetconfig -v
- check MIC if is in ready state
    micctrl -s
- if MIC is not in ready state then reset it
    micctrl -rw
- check MIC if is in ready state
    micctrl -s
- if MIC cannot be in ready state then you need to flash it (micflash -update) and reboot the host, and reset the MIC (it is in ready state (micctrl -s)
- once the MIC is in ready state then bring up MPSS:
    systemctl start mpss
  you can monitor the progress of the startup: dmesg | tail
- once the MIC is booted, you can ssh to it:
    ssh mic0
- for physical condition and information about the MIC:
    micinfo
- enable MPSS service, so that the MPSS is up after each reboot. Assuming you won't make changes to the MIC kernel imag
    systemctl enable mpss

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

C Appendix Troubleshootig the CUDA Configuration

CUDA has been configured on the HPMC. The following environment variables are already exported by default.

```
export LPATH=$LPATH:/usr/lib/nvidia:/opt/bin:/opt/lib64:/opt/lib:/usr/local/cuda/lib64
export LIBRARY_PATH=$LIBRARY_PATH:/usr/lib/nvidia:/opt/lib64:/opt/lib:/usr/local/cuda/lib64
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib/nvidia:/opt/lib64:/opt/lib:/usr/local/cuda/lib64
export PATH=$PATH:/opt/bin:/opt/lib64:/opt/lib:/usr/local/cuda/bin:/usr/local/cuda/lib64
```

If you have issues running CUDA related applications, make sure that the following shell script is available /etc/ccube/st/cu9sysenv.sh

```
export LPATH=$LPATH:/usr/lib64/openmpi/lib:/usr/lib/nvidia:/opt/bin:/opt/lib64:/opt/lib:/usr/local/cuda-9.2/l
export LIBRARY_PATH=$LIBRARY_PATH:/usr/lib64/openmpi/lib:/usr/lib/nvidia:/opt/lib64:/opt/lib:/usr/local/cuda-

export LUA_PATH='/cuda9/.luarocks/share/lua/5.1/?.lua;
/cuda9/.luarocks/share/lua/5.1/?/init.lua;
/cuda9/torch/install/share/lua/5.1/?.lua;
/cuda9/torch/install/share/lua/5.1/?/init.lua;
./?.lua;/cuda9/torch/install/share/luajit-2.1.0-beta1/?.lua;
/usr/local/share/lua/5.1/?.lua;/usr/local/share/lua/5.1/?/init.lua'

export LUA_CPATH='/cuda9/.luarocks/lib/lua/5.1/?.so;
/cuda9/torch/install/lib/lua/5.1/?.so;
/cuda9/torch/install/lib/?.so;./?.so;
/usr/local/lib/lua/5.1/?.so;/usr/local/lib/lua/5.1/loadall.so'

export PATH=$PATH:/usr/local/cuda-9.2/bin:/usr/local/cuda-9.2/lib64:/cuda9/CuViews:
/cuda9/CuExamples:/cuda9/CuExamples/comp:/cuda9/torch/install/bin:
/tools:/usr/local/ccube/bin:/usr/local/pgplot:$JPATH/bin

export LD_LIBRARY_PATH="${LD_LIBRARY_PATH:+$LD_LIBRARY_PATH:}/cuda9/torch/install/lib:
/usr/local/lib64:/usr/lib64/openmpi/lib:/usr/local/cuda-9.2/lib64:
/usr/local/cuda-9.2/libnvvp:/usr/local/cuda-9.2/libnsight:$JPATH/bin:$JPATH/jre/bin/classic"

export DYLD_LIBRARY_PATH=/cuda9/torch/install/lib

export LUA_CPATH='/cuda9/torch/install/lib/?.so;'$LUA_CPATH

# test CUDNN with layer.lua and bench.lua
export CUDNN_PATH=/usr/local/cuda-9.2/lib64/libcudnn.so.7.1.4

# NOTE the following can be overriden in user .bashrc or .profile in case using /mm03fs/miclib17, also source
#    but also need to change sym link of /opt/intel and may change /var/mpss/mic0/~/.bashrc
#export SINK_LD_LIBRARY_PATH=/mm03fs/miclib13
export SINK_LD_LIBRARY_PATH=/mc00/miclib13
```

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

D Shared Libraries of an Offloaded Program

Use readelf to locate the shared libraries for a program. For example, the following command prints the shared libraries needed by the program trap_offload.

```
(uos/HPMC9) 03:22 root@mic0: ~ # readelf -d trap_offload %%%pine grep NEEDED
```

```
(HPMC9) 03:22 root@mic0: ~ #  readelf -d trap_offload | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [libmpifort.so.12]
 0x0000000000000001 (NEEDED)             Shared library: [libmpi.so.12]
 0x0000000000000001 (NEEDED)             Shared library: [libdl.so.2]
 0x0000000000000001 (NEEDED)             Shared library: [librt.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libpthread.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libstdc++.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libm.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libiomp5.so]
 0x0000000000000001 (NEEDED)             Shared library: [liboffload.so.5]
 0x0000000000000001 (NEEDED)             Shared library: [libgcc_s.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

The following command prints the shared libraries needed by the shared library liboffload.so.5

```
(uos/HPMC9) 03:24 root@mic0: ~ # readelf -d liboffload.so.5 | grep NEEDED
```

```
(HPMC9) 03:24 root@mic0: ~ #  readelf -d liboffload.so.5 | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [libdl.so.2]
 0x0000000000000001 (NEEDED)             Shared library: [libimf.so]
 0x0000000000000001 (NEEDED)             Shared library: [libsvml.so]
 0x0000000000000001 (NEEDED)             Shared library: [libirng.so]
 0x0000000000000001 (NEEDED)             Shared library: [libm.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libiomp5.so]
 0x0000000000000001 (NEEDED)             Shared library: [libstdc++.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libgcc_s.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libintlc.so.5]
 0x0000000000000001 (NEEDED)             Shared library: [libpthread.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

Use readelf to locate the library runpath of a program. The following command prints the library runpath of the program trap_offload.

SAMPLE BOOK PREVIEW

```
(uos/HPMC9) 03:23 root@mic0: ~ # readelf -d trap_offload %%%pine grep RUNPATH
```

```
(HPMC9) 03:23 root@mic0: ~ #  readelf -d trap_offload | grep RUNPATH
<pre>
 0x000000000000001d (RUNPATH)            Library runpath: [/opt/intel/compilers_and_libraries_2017.7.259/linux/mpi/intel64/lib/release
```

Use objdump to locate the libraries needed by a program. The following command prints the libraries needed by the program trap_offload.

```
08:39 root@HPMC: /mm03fs/MPI # objdump -p trap_offload
   grep NEEDED
```

```
08:39 root@HPMC: /mm03fs/MPI #  objdump -p trap_offload | grep NEEDED
  NEEDED               libmpifort.so.12
  NEEDED               libmpi.so.12
  NEEDED               libdl.so.2
  NEEDED               librt.so.1
  NEEDED               libpthread.so.0
  NEEDED               libstdc++.so.6
  NEEDED               libm.so.6
  NEEDED               libiomp5.so
  NEEDED               liboffload.so.5
  NEEDED               libgcc_s.so.1
  NEEDED               libc.so.6
```

To disassemble programs compiled for the MIC, use the command objdump-mic. For example, the following command disassembles the program mic-gigaflops and generates the output file mic-gigaflops_xphi_Gg.ASMi

```
# objdump-mic mic-gigaflops
```

The command objdump-mic is the shell script shown below.

-LIS- Listing. D.1 [Shell script to disassemble Xeon Phi Native code using objdump]

```
1.     #!/bin/ksh
2.
3.     PROG=$1
4.     PHIg=$1
5.     PHIg+="_xphi_Gg"
6.     ASMi=$PHIg
7.     ASMi+=".ASMi"
8.     echo $PROG
9.     echo $PHIg
10.    echo $ASMi
11.
12.    scp $PHIg  mic0:/tmp
13.
14.    ssh mic0 "objdump -M intel -d /tmp/$PHIg  > /tmp/$ASMi"
15.
16.    scp mic0:/tmp/$ASMi  .
17.
```

SAMPLE BOOK PREVIEW

E A Simple OMP Program

Consider the following simple program to measure the DP GFLops by running many threads over OMP.

-LIS- Listing. E.1 [A simple OMP program gigaflop.cpp]

```
1.     #include <stdio.h>
2.     #include <stdlib.h>
3.     #include <omp.h>
4.
5.     unsigned int const SIZE=16;
6.     unsigned int const ITER=48000000;
7.
8.     extern double elapsedTime(void);
9.
10.    int main()
11.    {
12.        double startTime, duration;
13.        int i;
14.        __declspec(aligned(64)) double a[SIZE],b[SIZE],c[SIZE];
15.        // initialize
16.        for (i=0;i<SIZE;i++) {
17.            c[i]=b[i]=a[i]=(double)rand();
18.        }
19.        // warmup
20.        #pragma omp parallel for
21.        for(i=0;i<ITER;i++) {
22.            #pragma vector aligned(a,b,c)
23.            a[0:SIZE]=b[0:SIZE]*c[0:SIZE]+a[0:SIZE];
24.        }
25.        startTime = elapsedTime();
26.        #pragma omp parallel for
27.        for(i=0;i<ITER;i++) {
28.            #pragma vector aligned(a,b,c)
29.            a[0:SIZE]=b[0:SIZE]*c[0:SIZE]+a[0:SIZE];
30.        }
31.        duration = elapsedTime() - startTime;
32.        double Gflop = 2*SIZE*ITER/1e+9;
33.        double Gflops = Gflop/duration;
34.        printf("Running %d openmp threads\n", omp_get_max_threads());
35.        printf("DP GFlops = %f\n", Gflops);
36.        return 0;
37.    }
```

SAMPLE BOOK PREVIEW

Running the program on the HPMC's MIC coprocessor:

```
12:46 root@HPMC9: ~/coding #  xphi gigaflop_xphi

12:46 root@HPMC9: ~/coding #  xphi gigaflop_xphi
Running 224 openmp threads
DP GFlops = 154310202251.497009
```

Running the program on the HPMC host:

SAMPLE BOOK PREVIEW

F MPI Threads and Ranks and Context Switching

MIC Essentials:

Context switching over HPMC's MICs is not harmful, after couple of rounds of CW, they will clear up. The MIC has 240 cores, but what if I run the MPI program with 3000 threads. No problem. The Xeon Phi handles memory consistency inside its hardware and maintain cache coherency between all its active cores. When an RDMA engine reads or modifies data inside the MIC buffer, the hardware handles cache invalidations automatically across the ring bus. This can be seen in the excercise below where we monitored the context switching while displaying the interrupts with dint.

19:08 root@HPMC: /mm03fs/MPI # **OMP_NUM_THREADS=3000 I_MPI_DAPL_PROVIDER_LIST=ofa-v2-mlx4_0-1u I_MPI_PERHOST=1 I_MPI_DEBUG=5 I_MPI_FABRICS=shm:dapl I_MPI_FALLBACK=0 /opt/INTEL-XE-2017-update7/compilers_and_libraries_2017.7.259/linux/mpi/intel64/bin/mpiexec.hydra -n 8 -hosts hpmc7,hpmc9 ./trap_offload**

```
19:08 root@HPMC: /mm03fs/MPI #  OMP_NUM_THREADS=3000  I_MPI_DAPL_PROVIDER_LIST=ofa-v2-mlx4_0-1u
     I_MPI_PERHOST=1 I_MPI_DEBUG=5 \
     I_MPI_FABRICS=shm:dapl I_MPI_FALLBACK=0 \
     /opt/INTEL-XE-2017-update7/compilers_and_libraries_2017.7.259/linux/mpi/intel64/bin/mpiexec.hydra \
     -n 8 -hosts hpmc7,hpmc9  ./trap_offload

[0] MPI startup(): Multi-threaded optimized library
[1] DAPL startup(): trying to open DAPL provider from I_MPI_DAPL_PROVIDER: ofa-v2-mlx4_0-1u
[5] DAPL startup(): trying to open DAPL provider from I_MPI_DAPL_PROVIDER: ofa-v2-mlx4_0-1u
[3] DAPL startup(): trying to open DAPL provider from I_MPI_DAPL_PROVIDER: ofa-v2-mlx4_0-1u
[7] DAPL startup(): trying to open DAPL provider from I_MPI_DAPL_PROVIDER: ofa-v2-mlx4_0-1u
[0] DAPL startup(): trying to open DAPL provider from I_MPI_DAPL_PROVIDER: ofa-v2-mlx4_0-1u
[2] DAPL startup(): trying to open DAPL provider from I_MPI_DAPL_PROVIDER: ofa-v2-mlx4_0-1u
[4] DAPL startup(): trying to open DAPL provider from I_MPI_DAPL_PROVIDER: ofa-v2-mlx4_0-1u
[6] DAPL startup(): trying to open DAPL provider from I_MPI_DAPL_PROVIDER: ofa-v2-mlx4_0-1u
[1] MPI startup(): DAPL provider ofa-v2-mlx4_0-1u
[1] MPI startup(): shm and dapl data transfer modes
[5] MPI startup(): DAPL provider ofa-v2-mlx4_0-1u
[5] MPI startup(): shm and dapl data transfer modes
[3] MPI startup(): DAPL provider ofa-v2-mlx4_0-1u
[7] MPI startup(): DAPL provider ofa-v2-mlx4_0-1u
[3] MPI startup(): shm and dapl data transfer modes
[7] MPI startup(): shm and dapl data transfer modes
[0] MPI startup(): DAPL provider ofa-v2-mlx4_0-1u
[2] MPI startup(): DAPL provider ofa-v2-mlx4_0-1u
[4] MPI startup(): DAPL provider ofa-v2-mlx4_0-1u
[6] MPI startup(): DAPL provider ofa-v2-mlx4_0-1u
[0] MPI startup(): shm and dapl data transfer modes
[2] MPI startup(): shm and dapl data transfer modes
[4] MPI startup(): shm and dapl data transfer modes
[6] MPI startup(): shm and dapl data transfer modes
[1] MPID_nem_init_dapl_coll_fns(): User set DAPL collective mask = 0000
[1] MPID_nem_init_dapl_coll_fns(): Effective DAPL collective mask = 0000
[3] MPID_nem_init_dapl_coll_fns(): User set DAPL collective mask = 0000
[3] MPID_nem_init_dapl_coll_fns(): Effective DAPL collective mask = 0000
[5] MPID_nem_init_dapl_coll_fns(): User set DAPL collective mask = 0000
[5] MPID_nem_init_dapl_coll_fns(): Effective DAPL collective mask = 0000
[7] MPID_nem_init_dapl_coll_fns(): User set DAPL collective mask = 0000
[7] MPID_nem_init_dapl_coll_fns(): Effective DAPL collective mask = 0000
[0] MPID_nem_init_dapl_coll_fns(): User set DAPL collective mask = 0000
[0] MPID_nem_init_dapl_coll_fns(): Effective DAPL collective mask = 0000
[2] MPID_nem_init_dapl_coll_fns(): User set DAPL collective mask = 0000
[2] MPID_nem_init_dapl_coll_fns(): Effective DAPL collective mask = 0000
[4] MPID_nem_init_dapl_coll_fns(): User set DAPL collective mask = 0000
[4] MPID_nem_init_dapl_coll_fns(): Effective DAPL collective mask = 0000
[6] MPID_nem_init_dapl_coll_fns(): User set DAPL collective mask = 0000
[6] MPID_nem_init_dapl_coll_fns(): Effective DAPL collective mask = 0000
[0] MPI startup(): Rank    Pid      Node name  Pin cpu
[0] MPI startup(): 0       25948    HPMC7      {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,68,69,70,71,72,73,74,75,76,77,78
                                    81,82,83,84,136,137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,152,
                                    204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219,220}
[0] MPI startup(): 1       29906    HPMC9      {0,1,2,3,4,5,6,7,8,9,10,11,12,13,56,57,58,59,60,61,62,63,64,65,66,67,68,69
[0] MPI startup(): 2       25949    HPMC7      {17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,85,86,87,88,89,90,91,9
                                    ,95,96,97,98,99,100,101,153,154,155,156,157,158,159,160,161,162,163,164,165,166,
                                    167,168,169,221,222,223,224,225,226,227,228,229,230,231,232,233,234,235,236,237}
[0] MPI startup(): 3       29907    HPMC9      {14,15,16,17,18,19,20,21,22,23,24,25,26,27,70,71,72,73,74,75,76,77,78,79,8
                                    ,83}
[0] MPI startup(): 4       25950    HPMC7      {34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,102,103,104,105,106,10
                                    09,110,111,112,113,114,115,116,117,118,170,171,172,173,174,175,176,177,178,179,1
                                    80,181,182,183,184,185,186,238,239,240,241,242,243,244,245,246,247,248,249,250,2
                                    51,252,253,254}
[0] MPI startup(): 5       29908    HPMC9      {28,29,30,31,32,33,34,35,36,37,38,39,40,41,84,85,86,87,88,89,90,91,92,93,9
                                    ,97}
[0] MPI startup(): 6       25951    HPMC7      {51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,119,120,121,122,123,12
                                    26,127,128,129,130,131,132,133,134,135,187,188,189,190,191,192,193,194,195,196,1
                                    97,198,199,200,201,202,203,255,256,257,258,259,260,261,262,263,264,265,266,267,2
                                    68,269,270,271}
[0] MPI startup(): 7       29909    HPMC9      {42,43,44,45,46,47,48,49,50,51,52,53,54,55,98,99,100,101,102,103,104,105,1
                                    108,109,110,111}
[0] MPI startup(): I_MPI_DAPL_PROVIDER_LIST=ofa-v2-mlx4_0-1u
[0] MPI startup(): I_MPI_DEBUG=5
[0] MPI startup(): I_MPI_FABRICS=shm:dapl
[0] MPI startup(): I_MPI_FALLBACK=0
[0] MPI startup(): I_MPI_INFO_NUMA_NODE_MAP=qib0:0,mic0:0
[0] MPI startup(): I_MPI_INFO_NUMA_NODE_NUM=2
[0] MPI startup(): I_MPI_PIN_MAPPING=4:0 0,2 17,4 34,6 51
rank 1 of 8 on HPMC9: 12.618342 seconds
```

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

G dint: display the system interrupts

Your HPMC comes with the dint program that display the system interrupts. dint is a command that you can issue at the command prompt. Since your HPMC has hundreds of cores, and your Xeon Phi MIC has as well hundreds of cores, you need a utility that can display the system interrupts while focusing on the busiest cores.

The chapter "Programming and Running by MIC Native Loading" showed how to run diffusion_omp by offloading it to the MIC. In the following we show how to use the dint command to display the interrupts as the program is executing.

The interrupt names may be different for each system. To display the names of the interrupts, you can simply use dint with the -info option:

```
# dint -info
```

dint can display all cores. Use -transpose option so that the 228 cores are displayed one per row, a neater view.

```
(uos/HPMC9) 03:59 root@mic0: /tools # ./dint -transpose -iter 7 -delay 1 -interrupts
LOC,RES,CAL,4,5,26,TLB
```

SAMPLE BOOK PREVIEW

Use the -prune option to display the cores by pruning a range between the beginning and the end:

```
(uos/HPMC9) 03:59 root@mic0: /tools # ./dint -transpose -iter 7 -delay 1 -interrupts
LOC,RES,CAL,4,5,26,TLB -prune 3,5
```

Displaying all the cores can be distracting, so to focus only on a range of cores, just add in -sortby option:

```
(uos/HPMC9) 04:02 root@mic0: /tools # ./dint -transpose -iter 7 -delay 1 -interrupts
LOC,RES,CAL,4,5,26,TLB -prune 3,5 -sortby RES
```

To print all cores focusing on the RES interrupt, use this command:

```
(uos/HPMC9) 03:59 root@mic0: /tools # ./dint -transpose -iter 7 -delay 1 -interrupts
LOC,RES,CAL,4,5,26,TLB -sortby RES
```

Use the -transpose option so that each core or numbered-CPU is displayed on a row.

The following shows the dint command running on the MIC of hpmc9, where we focused on five cores that are busy with the RES (Rescheduling interrupts). After each iteration the busiest cores with the RES are displayed first. Notice how after each iteration the header shows only these CPU's with the busiest RES.

```
(uos/HPMC9) 03:40 root@mic0: /tools # ./dint -NVU 5 -iter 7 -delay 1 -interrupts LOC,RES,CAL,4,5,26,TLB -
sortby RES
```

```
(HPMC9) 03:40 root@mic0: /tools #  ./dint -NVU 5 -iter 7 -delay 1 -interrupts LOC,RES,CAL,4,5,26,TLB -sortby RE
                          CPU227      CPU0    CPU225    CPU167     CPU26    summed
2026-04-23 03:41:04     LOC        70       121        80       162       162       595     Local timer interru
2026-04-23 03:41:04     RES        12         8         3         2         2        27     Rescheduling interr
2026-04-23 03:41:04     CAL        84         0        47         0         0       131     Function call inter
2026-04-23 03:41:04       4         0         0         0         0         0         0     IO-APIC-edge      c
2026-04-23 03:41:04       5         0         0         0         0         0         0     IO-APIC-edge      c
2026-04-23 03:41:04      26         0         0         0         0         0         0     SBOX-ICR-edge
2026-04-23 03:41:04     TLB         0         0         1         0         0         1     TLB shootdowns

                          CPU226    CPU227      CPU0    CPU225    CPU119    summed
2026-04-23 03:41:06     LOC       118       132       150        62       159       621     Local timer interru
2026-04-23 03:41:06     RES        12         9         8         7         3        39     Rescheduling interr
2026-04-23 03:41:06     CAL         0         4         8        93         0       105     Function call inter
```

SAMPLE BOOK PREVIEW

H dcpus: display the system stats

Your HPMC comes with the dcpus program that display the system stats. dcpus is a command that you issue at the command prompt. Since your HPMC has hundreds of cores, and your Xeon Phi MIC has as well hundreds of cores, you need a utility that can display the system stats while focusing on the busiest cores.

The chapter "Programming and Running by MIC Native Loading" showed how to run diffusion_omp by offloading it to the MIC. In the following we show how to use the dcpus command to display the stat of the cores as the program is executing.

To display the stats names on your system, you can simply use dcpus with the -info option:

```
# dcpus -info
```

dcpus shows that the HPMC7 has 272 CPU cores.

```
01:56 root@HPMC7: ~ #  dcpus -info

01:56 root@HPMC7: ~ #  dcpus -info
  SYSTEM TICKS 100

  CPU COUNT 272

  STAT MNEMONICS: user nice system idle iowait irq softirq steal guest guestnice

        user    Time spent with normal processing in user mode
        nice    Time spent with niced processes in user mode
      system    Time spent running in kernel mode
        idle    Time spent in vacations twiddling thumbs
      iowait    Time spent waiting for I/O to completed. This is considered idle time.
         irq    Time spent serving hardware interrupts
     softirq    Time spent serving software interrupts
       steal    Time stolen by other operating systems running in a virtual environment
       guest    Time spent for running a virtual CPU or guest OS under the control of the kernel
    guestnice
```

SAMPLE BOOK PREVIEW

The first MIC hosted by HPMC7 is mic0, the following command shows that it has 244 CPU cores.

```
01:57 root@HPMC7: ~ #  ssh mic0 "/tools/dcpus -info"

01:57 root@HPMC7: ~ #  ssh mic0 "/tools/dcpus -info"
  SYSTEM TICKS 100

  CPU COUNT 244

  STAT MNEMONICS: user nice system idle iowait irq softirq steal guest guestnice

        user    Time spent with normal processing in user mode
        nice    Time spent with niced processes in user mode
      system    Time spent running in kernel mode
        idle    Time spent in vacations twiddling thumbs
      iowait    Time spent waiting for I/O to completed. This is considered idle time.
         irq    Time spent serving hardware interrupts
     softirq    Time spent serving software interrupts
       steal    Time stolen by other operating systems running in a virtual environment
       guest    Time spent for running a virtual CPU or guest OS under the control of the kernel
    guestnice

  THE CACHEDIR IS /tmp/cached_sysstats
```

dcpus can display all cores. Use -transpose option so that the hundreds of cores are displayed one per row - a neater view.

```
(uos/HPMC9) 03:59 root@mic0: /tools #  ./dcpus -transpose -iter 7 -delay 1
```

Use the -prune option to display the cores by pruning a range between the beginning and the end.

```
(uos/HPMC9) 03:59 root@mic0: /tools #  ./dcpus -transpose -iter 7 -delay 1 -prune 3,5
```

Displaying all the cores can be distracting, so to focus only on a range of cores where a specific stat name is increasing, just add in -sortby option. The following focus on the system stat, showing only the topmost three cores and the last five cores.

SAMPLE BOOK PREVIEW

I gpdata3: plot 2D and 3D graphs

To plot performance graphs we use gpdata3. The following commands show how we plot the performance graphs shown in this book.

Performance of matrix multiplication on the GPU.

gpdata3 DP-FMA,3:11wlp-lt-1-lw-1.9-pt-4-ps-1.7-rgb-red DP-noFMA,3:11wlp-lt-1-lwp-1.9-pt-5-ps-1.7-rgb-red SP-FMA,3:11wlp-lt-1-lw-1.9-pt-6-ps-1.7-rgb-green SP-noFMA,3:11wlp-lt-1-lwp-1.9-pt-7-ps-1.7-rgb-green -title "Matrix Multiplication on GPU" -xlabel "Matrix Size" -ylabel "GFlops"

```
# gpdata3 DP-FMA,3:11wlp-lt-1-lw-1.9-pt-4-ps-1.7-rgb-red \
          DP-noFMA,3:11wlp-lt-1-lwp-1.9-pt-5-ps-1.7-rgb-red \
          SP-FMA,3:11wlp-lt-1-lw-1.9-pt-6-ps-1.7-rgb-green \
          SP-noFMA,3:11wlp-lt-1-lwp-1.9-pt-7-ps-1.7-rgb-green \
          -title "Matrix Multiplication on GPU" \
          -xlabel "Matrix Size" -ylabel "GFlops"
```

Performance of matrix multiplication on the MIC.

SAMPLE BOOK PREVIEW

gpdata3 MIC-DP,5:13wlp-lt-1-lw-1.9-pt-4-ps-1.7-rgb-red MIC-SP,5:13wlp-lt-1-lwp-1.9-pt-5-ps-1.7-rgb-red -title "Matrix Multiplication on MIC Using 240 Threads" -xlabel "Matrix Size" -ylabel "GFlops"

```
# gpdata3 MIC-DP,5:13wlp-lt-1-lw-1.9-pt-4-ps-1.7-rgb-red \
          MIC-SP,5:13wlp-lt-1-lwp-1.9-pt-5-ps-1.7-rgb-red \
         -title "Matrix Multiplication on MIC Using 240 Threads" \
         -xlabel "Matrix Size" -ylabel "GFlops"
```

Compare matrix multiplication with double precision operations on GPU vs MIC. GPU performance is better than the MIC, however it is limited to matrix size 500x500, while on MIC it can be 1000x1000 and even higher, as it is managed by OpenMP.

gpdata3 MIC-DP,5:13wlp-lt-1-lw-1.9-pt-4-ps-1.7-rgb-blue DP-FMA,3:11wlp-lt-1-lw-1.9-pt-2-ps-1.7-rgb-red -title "Matrix Multiplication on GPU and on MIC Using 240 Threads" -xlabel "Matrix Size" -ylabel "GFlops"

```
gpdata3 MIC-DP,5:13wlp-lt-1-lw-1.9-pt-4-ps-1.7-rgb-blue \
     DP-FMA,3:11wlp-lt-1-lw-1.9-pt-2-ps-1.7-rgb-red \
     -title "Matrix Multiplication on GPU and on MIC Using 240 Threads" \
     -xlabel "Matrix Size" -ylabel "GFlops"
```

SAMPLE BOOK PREVIEW

J How to display the system interrupts of your HPMC

Use the dint command to display the interrupts of your HPMC computer.

17:31 root@HPMC7: ~ # **dint -info**

```
17:31 root@HPMC7: ~ #  dint -info
  CPU COUNT 272

     0    IR-IO-APIC-edge      timer
     8    IR-IO-APIC-edge      rtc0
     9    IR-IO-APIC-fasteoi   acpi
    18    IR-IO-APIC-fasteoi   ehci_hcd:usb1, ehci_hcd:usb2, i801_smbus
    25    IR-PCI-MSI-edge      PCIe PME
    27    IR-PCI-MSI-edge      PCIe PME
    29    IR-PCI-MSI-edge      PCIe PME
    30    IR-PCI-MSI-edge      PCIe PME
    31    IR-PCI-MSI-edge      PCIe PME
    32    IR-PCI-MSI-edge      xhci_hcd
    33    IR-PCI-MSI-edge      0000:00:11.4
    34    IR-PCI-MSI-edge      mic
    35    IR-PCI-MSI-edge      ib_qib0
    36    IR-PCI-MSI-edge      ib_qib0 (buf avail)
    37    IR-PCI-MSI-edge      ib_qib0 (sdma 0)
    38    IR-PCI-MSI-edge      ib_qib0 (sdmaI 0)
    39    IR-PCI-MSI-edge      ib_qib0 (sdmaP 0)
    40    IR-PCI-MSI-edge      ib_qib0 (sdmaC 0)
    41    IR-PCI-MSI-edge      ib_qib0 (kctx)
    42    IR-IO-APIC-fasteoi   snd_hda_intel
    43    IR-PCI-MSI-edge      enp5s0f0
    44    IR-PCI-MSI-edge      enp5s0f0-TxRx-0
    45    IR-PCI-MSI-edge      enp5s0f0-TxRx-1
    46    IR-PCI-MSI-edge      enp5s0f0-TxRx-2
    47    IR-PCI-MSI-edge      enp5s0f0-TxRx-3
    48    DMAR_MSI-edge      dmar0
    49    IR-PCI-MSI-edge      enp5s0f0-TxRx-4
    50    IR-PCI-MSI-edge      enp5s0f0-TxRx-5
    51    IR-PCI-MSI-edge      enp5s0f0-TxRx-6
    52    IR-PCI-MSI-edge      enp5s0f0-TxRx-7
    53    IR-PCI-MSI-edge      enp5s0f1
    54    IR-PCI-MSI-edge      enp5s0f1-TxRx-0
    55    IR-PCI-MSI-edge      enp5s0f1-TxRx-1
    56    IR-PCI-MSI-edge      enp5s0f1-TxRx-2
    57    IR-PCI-MSI-edge      enp5s0f1-TxRx-3
    58    IR-PCI-MSI-edge      enp5s0f1-TxRx-4
    59    IR-PCI-MSI-edge      enp5s0f1-TxRx-5
    60    IR-PCI-MSI-edge      enp5s0f1-TxRx-6
    61    IR-PCI-MSI-edge      enp5s0f1-TxRx-7
  NMI    Non-maskable interrupts
  LOC    Local timer interrupts
  SPU    Spurious interrupts
  PMI    Performance monitoring interrupts
  IWI    IRQ work interrupts
  RTR    APIC ICR read retries
  RES    Rescheduling interrupts
  CAL    Function call interrupts
  TLB    TLB shootdowns
  TRM    Thermal event interrupts
  THR    Threshold APIC interrupts
  DFR    Deferred Error APIC interrupts
  MCE    Machine check exceptions
  MCP    Machine check polls
  PIN    Posted-interrupt notification event
  PIW    Posted-interrupt wakeup event
```

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

K How to display the interrupts of the MIC

To display the interrupts of the Xeon Phi MIC, ssh to the MIC and issue the dint command.

(uos/HPMC7) 17:50 root@mic0: /tools # **./dint -info**

```
(HPMC7) 17:50 root@mic0: /tools #  ./dint -info
  CPU COUNT 228

  INTERRUPTS MNEMONICS: 4 5 6 7 13 17 26 27 28 30 NMI LOC SPU PMI IWI RES CAL TLB TRM THR MCE MCP

     4    IO-APIC-edge      dma channel
     5    IO-APIC-edge      dma channel
     6    IO-APIC-edge      dma channel
     7    IO-APIC-edge      dma channel
    13    IO-APIC-edge      TMU
    17    IO-APIC-edge      SCIF INTR 0
    26    SBOX-ICR-edge      vnet intr
    27    SBOX-ICR-edge      Shutdown
    28    SBOX-ICR-edge      hvc intr
    30    SBOX-ICR-edge      PC35_Wakeup
   NMI    Non-maskable interrupts
   LOC    Local timer interrupts
   SPU    Spurious interrupts
   PMI    Performance monitoring interrupts
   IWI    IRQ work interrupts
   RES    Rescheduling interrupts
   CAL    Function call interrupts
   TLB    TLB shootdowns
   TRM    Thermal event interrupts
   THR    Threshold APIC interrupts
   MCE    Machine check exceptions
   MCP    Machine check polls
```

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

L FLOPS = Floating-point Operations per Second

FLOPS is the number of floating-point arithmetic on real numbers a processor can perform in one second.

FLOPS is measured in scaling tiers denoted as follows:

```
FLOPS: Floating Point Operations Per Second
KFLOPS: Kilo (thousand) FLOPS - 10³  or 1000  FLOPS
MFLOPS: Mega (million) FLOPS  - 10⁶  or 1000000  FLOPS
GFLOPS: Giga (billion) FLOPS  - 10⁹  or 1000000000  FLOPS
TFLOPS: Tera (trillion) FLOPS - 10¹² or 1000000000000  FLOPS
```

Over the last several decades, the Floating-Point Operations Per Second (FLOPS) of computers has grown exponentially.

The 1954's IBM 704 with its vacuum-tube logic circuitry, 36-bit words, magnetic-core memory, and hardware floating-point arithmetic, executed each operation in 240 microseconds, which is roughly 4,000 multiplications or divisions per second. The IBM 704 executed up to 12,000 floating-point additions per second or 12 KFLOPS. The 1965's 12-bit minicomputer PDP-8 does not have native hardware support for floating-point math. To compute fractional or large numbers, it had to rely on a multi-word software library. The PDP-8 has a floating-point performance of approximately 1,000 to 2,000 FLOPS or 1 to 2 KFLOPS.

SAMPLE BOOK PREVIEW

The 1976's DEC PDP-11/34 minicomputer, when equipped with hardware-assisted floating-point units, it has a floating-point performance of approximately 50,000 FLOPS or 50 KFLOPS.



Fig. L.1

HPMC 2026 by Bassem Jamaleddine

The 1980's Intel 8087 math coprocessor performs approximately 50,000 FLOPS or 50 KFOPS.

The 1981's IBM 3081 processor complex delivered roughly 10 to 12 million instructions per second (MIPS) (or about 5 to 7 MIPS per central processor unit. When operated across its dyadic (dual-CPU) it delivered in the range of 10–12 MIPS. An individual IBM 3081 models performed around 3,000,000 floating-point operations per second or 3 MFLOPS.

The 1975's Cray-1 supercomputer with a processor operating at an 80 MHz clock frequency (12.5-nanosecond clock cycle), has a peak processing speed of 160,000,000 FLOPS or 160 MFLOPS.

The Intel Xeon Phi Knights Corner coprocessor (with its vector units) is rated by its math

SAMPLE BOOK PREVIEW

M Installing Torch cuDNN and PyTorch

To install cuDNN cudnn.torch, the Torch7 bindings for NVIDIA's cuDNN library, both cutorch and cunn must be installed first.

Before installing Lua/Torch the NVIDIA library must be exported in LD_LIBRARY_PATH. To install cudnn.torch (the Torch7 bindings for NVIDIA's cuDNN library),

```
# export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/lib/nvidia:/opt/lib64:/opt/lib:/usr/local/cuda/lib64:/usr/lib64/atlas
```

To install Torch, in a terminal, run the commands

```
# curl -sk https://raw.githubusercontent.com/torch/ezinstall/master/install-deps
  bash

# git clone https://github.com/torch/distro.git ~/torch --recursive

# cd ~/torch

# ./install.sh
```

If you want to add the Torch environment variables to your profile or to the system profile, you can find these variables in cupath.sh.

```
# cat cupath.sh
```

To install cutorch (the CUDA backend for Torch7) using LuaRocks:

```
# luarocks install --server=https://luarocks.org/dev cutorch
```

To install cunn, the CUDA-enabled neural network library for Torch7

```
# luarocks install cunn
```

SAMPLE BOOK PREVIEW

Installing PyTorch requires to have cmake3 installed, python3 with openssl enabled, and the specific version of GCC that is compatible with pytorch distribution. The following shows how to install PyTorch on your HPMC.

SAMPLE BOOK PREVIEW

First install cmake3

```
# yum install cmake3
```

If cmake3 failed to install, then you can install it explicitly by downloading the package and using the rpm command.

```
# mkdir yumdown-cmake3

# yumdownloader --destdir=./yumdown-cmake3 --resolve cmake3

# (cd yumdown-cmake3; rpm -qilp cmake3-data-3.17.5-1.el7.noarch.rpm)



# cd yumdown-cmake3/cmake3

# rpm2cpio ../cmake3-3.17.5-1.el7.x86_64.rpm | cpio -idmv

# cd ..

# cd cmake3-data

# rpm2cpio ../cmake3-data-3.17.5-1.el7.noarch.rpm | cpio -idmv
```

Now that the rpm files are extracted in the current directory, you can install them with the rpm command:

```
# rpm -i --nodeps *.rpm
```

Test if openssl can be loaded from python

```
# python3

Python 3.8.1 (default, Aug  3 2022, 13:10:13)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import socket
>>> hasattr(socket,"SSL")
False
>>> hasattr(socket,"OPENSSL")
False
```

If openssl is not available in python, then install it and recompile python.

```
# yum install openssl

# yum install openssl-devel

# wget http://www.openssl.org/source/openssl-1.0.2e.tar.gz
  <<< INSTALL openssl from source
```

SAMPLE BOOK PREVIEW

N Historical Perspective on High Performance Switch

The Vulcan system was developed at the IBM T.J. Watson Research Center, started in 1988 to be completed in 1992. Vulcan was a system for an experimental massively parallel system of a scalable, distributed-memory parallel computer based on Intel i860 compute-nodes and a low-latency, high-bandwidth, multistage interconnection network. Vulcan consisted of 16 nodes connected by a network of switches that is very similar to the high-performance network used in the SP1. Viper was the operating system for Vulcan. Viper features task management facilities, multithreading, and point-to-point communication protocols. Viper was designed as a kernel for distributed-memory parallel computers. Venus is a prototype design of a collective communication library for massively parallel computers. Venus incorporates features such as Process Groups and Process Channels and supports coordination of and communication among processes in dynamic situations.

The AIX Parallel Environment supported the EUI message passing library by selecting a partition of the system that consisted of a subset of nodes. For example, each computation process is executed on one dedicated node.

In EUI the entire communication stack is executed in user space, hence bypassing the UNIX kernel overhead like system calls that intervene with the internal device drivers. Each node has its own communication adapter. Hence, EUI featured basic point-to-point and collective communication routines. With the pipe layer providing a reliable byte-stream oriented communication layer, eech processor maintained a send and a receive buffer (called pipe) to every other processor in the parallel job. These buffers evolved from UNIX pipes and sockets, yet they were placed in user space to avoid the cost of a UNIX kernel access: illiminating the CPU intervention to context switch every time a user-space application writes data to or reads data from a pipe. This layer uses the mechanisms of flow control, acknowledgments and retry after time-out etc. The EUI is implemented as a collection of C subroutines which are invoked from FORTRAN, C, and C++. From FORTRAN, the EUI is invoked as subprograms and arguments are passed by reference.
When the HPC community moved toward standardizing parallel code, the EUI (and its extended 1994 release EUI-H) served as one of communication layer that is used in putting the initial MPI specification. MPI-1 included many of the function calls found in EUI-H, sometimes recoding functions logic like mp_send() to MPI_Send(), and coding MPI_Bcast and MPI_Reduce based on EUI-H handling broadcast operations (one node sending data to all nodes) and reduction operations (gathering and sum/averaging data from all nodes), and adding Communicators MPI_COMM_WORLD for logical processor ranks.

SAMPLE Book PREVIEW

THE HPS SWITCH
In 1993, the SP1 High-Performance Switch (HPS) was the first commercialized Vulcan technology that interconnected individual RS/6000 processor nodes. This marked IBM’s true entry into distributed-memory parallel supercomputing.

SP1 was based on Micro Channel Architecture (MCA) that was invented primarily for the IBM PS/2 personal computers. While the SP1 switch chip was fast, it was bottlenecked by a passive communication adapter limited by the SP1 bus bandwidth. Because the CPU had to manage packet transfers by copying every single data packet byte-by-byte into the adapter's FIFO buffers over the Micro Channel bus, the effective network throughput was severely bottlenecked.
Each individual node connected to the HPS via a passive communication adapter card plugged into its Micro Channel slot. The adapter was relatively simple, consisting primarily of control registers and FIFO buffers for processing incoming and outgoing data packets. It lacked Direct Memory Access (DMA) capabilities.
To fix the performance issue of the communication adapter card, and to decouple it from the CPU, IBM didn't need to reinvent the wheel. The solution was there in the SP/2 Micro Channel Architecture (MCA).
IBM SP/2 and its OS/2 may not have been broadly selling to consumers, but SP/2 MCA had a native decentralized DMA protocol mechanism that IBM could use and move forward with delivering to deliver Massively Parallel Processing (MPP) using clustered IBM's RS/6000 office workstations.

SAMPLE BOOK PREVIEW

O Historical Perspective on Multiply Add Fused

One of the most common operation in matrix computations is to compute d = a * b + c on floating point data. The operations involved in getting d are: multiply, normalize, round, normalize, add, normalize, round.

The floating point multiplication, under IEEE 754 standard, requires to normalize the data before rounding it. We normalize the result so that the binary point is in the correct position, and if an overflow shift or left shift happens, we adjust the exponent accordingly. Next, we round the normalized result to fit into the floating point targetted precision.

It is possible to avoid the intermediate rounding operation, and even get more precision, by having an instruction **(multiply-add fused)** that does: multiply, normalize, add, normalize, round.

In 1990, Robert K. Montoye of IBM introduced the fused multiply-add and referred to it as **"multiply-add fused"**, for its IBM RS/6000 POWER1 workstation. Montoye filed the foundational U.S. Patent for this architecture: "Floating Point Unit for Calculating A=XY+Z Having Simultaneous Multiply and Add", under U.S. Patent No. 4,969,118 which was granted in November 1990. The concept was so effective that it was standardized across the industry in the IEEE 754-2008 floating-point specification.

SAMPLE BOOK PREVIEW

In 1992, the PowerPC 601 processor powered the IBM RS/6000 workstations. The 601 contains 2.8 million transistors, has a 32 KB unified L1 cache, supports an external L2 cache, and can process up to 3 instructions per cycle. This RISC processor outperformed Intelâ€™s Pentium. IBM released an updated version 601+ with higher speeds from 90 MHz to 120 MHz, running at 2x or 3x bus speed.
The 601 processor included the **Multiply-Add Fused (MAF) instruction** as one of the most powerful features of the IBM RS/6000. It allows the execution unit to multiply two operands, add a third operand, and round the result only once at the very end. This yields higher performance and greater floating-point precision than evaluating the operations separately.

The PowerPC 601 architecture supports two primary sets of assembly mnemonics for these operations depending on whether you are compiling for single or double precision:

The following table shows the fused operations and their mnemonics on PowerPC 601.

Operation Type	Double Precision Mnemonic	Single Precision Mnemonic	Mathematical Formula
Multiply-Add	fmadd or fma	fmadds	fr4 = (fr1 * fr2) + fr3
Multiply-Subtract	fmsub or fms	fmsubs	fr4 = (fr1 * fr2) - fr3
Negative Multiply-Add	fnmadd or fnma	fnmadds	fr4 = -((fr1 * fr2) + fr3)

Tab. O.1 [Fused operations and their mnemonics on PowerPC 601]

```
# Code Example: Fused Multiply-Add (Double Precision)
# Goal: Compute Result = (X * Y) + Z
# Registers utilized:
#   fr1 (X), fr2 (Y), fr3 (Z), fr4 (Result)
#   fr1 (A), fr2 (B), fr3 (C), fr4 (Result)

.globl fused_calculation
.csect .fused_calculation[PR]
```

SAMPLE BOOK PREVIEW

P Xeon Phi and MIC versus GPU

Table L1 summarized the differences between the HPMC's Xeon Phi and its MIC, versus GPU. Both the HPMC's mainboard and the MIC have Xeon Phi processors, and run a Linux operating system. Therefore, you will find most of the UNIX computer organization architecture concepts applied to them. This is not true for the GPU.

Intel Xeon Phi MIC (Many Integrated Core)	GPU (Graphics Processing Unit)
Intel Xeon MIC cores are actual, full-fledged lightweight x86 CPU cores with wide SIMD vector units.	GPU cores are non-x86 stream processors and cannot be used for general purpose execution.
Can be programmed with native C, C++, Fortran using OpenMP, MPI, and x86 SIMD intrinsics.	CUDA or OpenCL specialized languages utilizing streaming multiprocessor grids.
You can compile and run standard C, C++, or Fortran code directly on an Intel MIC using familiar CPU toolchains.	GPUs require specialized programming models like Nvidia's CUDA or OpenCL.
MIC x86 cores share hardware cache coherently.	GPUs require separate memory management frameworks.
The MIC runs a stripped-down version of Linux and allows users to ssh directly into the card and run UNIX administrative commands.	GPUs operate strictly as slave accelerator cards managed by the host OS and CPU.
MIC users can ssh into the card and debug with GDB.	Debugging CUDA with cuda-gdb is clumsy, slow, and mentally taxing compared to standard CPU debugging.
Intel MIC perform better on regular data-access patterns or double-precision arithmetic.	GPUs pull ahead on high-throughput and massive-scale parallel operations, at the cost of precision arithmetic.
The RDMA implementation is direct on InfiniBand verbs, and SCIF integration over standard network layer.	Specialized GPUDirect RDMA or peer-to-peer PCIe mapping frameworks.
Memory caching is fully cache-coherent L1/L2 networks across many x86 cores.	Non-coherent and is highly managed hierarchical structures optimized for massive throughput.
Standard MPI implementations run natively on the card, enabling the system to trigger an RDMA operation directly out of its localized memory footprint using standard verbs.	A GPU cannot act as an independent network participant; it requires the host system to register GPU device memory pages through pinned memory structures, routing network requests across a dedicated PCIe layout to expose the buffer directly to a physical Host Channel Adapter (HCA
Xeon Phi maintains hardware cache coherency between all active x86 processing cores inside its hardware mesh using Distributed Tag Directory (DTD), where every cache line maps to a specific directory agent on the ring bus(*) based on its address hash. When an RDMA engine reads or modifies data inside a Xeon MIC buffer, the hardware handles cache invalidations automatically across the ring bus.	Unlike the MIC, the burden of cache management is offloaded onto the developers. Since GPUs cannot act as an independent network participant; it requires the host system to manage the data synchronization onto the execution software stack before and after an RDMA event takes place.

SAMPLE BOOK PREVIEW

Tab. P.1 [Xeon Phi / MIC vs GPU]

SAMPLE BOOK PREVIEW

Glossary

ALU The Xeon Phi VPU contain standard scalar ALUs for basic control flow and integer operations, but also the core execution engine of the Xeon Phi VPU is a massive, dual-issue Vector Processing Unit that perform 512-bit wide Vector ALU integrated directly into each core's VPU..

ATLAS ATLAS (Automatically Tuned Linear Algebra Software), is an empirical tuning system that produces a BLAS (Basic Linear Algebra Subprograms) library which has been specifically optimized for the platform you install ATLAS on. The BLAS are a set of building block routines which, when tuned well, allow more complicated Linear Algebra operations such as solving linear equations or finding eigenvalues to run extremely efficiently (this is important, since these operations are computationally intensive). ATLAS also natively provides a few routines from the LAPACK (Linear Algebra PACKage). LAPACK is an extremely comprehensive FORTRAN package for solving the most commonly occurring problems in numerical linear algebra.

AVX Advanced Vector Extensions (AVX) are extensions to the x86 instruction set architecture for the Intel Xeon Phi microprocessors that perform Single Instruction, Multiple Data (SIMD) operations to speed up computations of heavy workloads. The AVX-512F (Advanced Vector Extensions 512 Foundation), supported across Intel Xeon Phi, is the core instruction set foundation for 512-bit single-instruction multiple-data (SIMD) operations in x86 processors. It has 32 registers (ZMM0–ZMM31), each 512 bits wide, and 8 dedicated opmask registers (k0–k7) for conditional execution. It processes up to 32 double-precision (SP) or 64 single-precision (DP) floating-point values per clock cycle using dual 512-bit FMA (Fused Multiply/Add) units. It introduced the EVEX prefix, variable masking (merge or zero-masking), broadcasting, and embedded rounding. The grep command on the cpuinfo of the HPMC with the Knight Landing reported: avx avx2 avx512f avx512pf avx512er avx512cd.

SAMPLE BOOK PREVIEW

BMIN BMIN stands for a Bidirectional Multistage Interconnection Network, a class of indirect network topology used in high-performance computing (HPC) and parallel computer architectures.

Caffe Caffe, Convolutional Architecture for Fast Feature Embedding, is a deep learning framework developed at University of California, Berkeley.

COI COI, coprocessor offload infrastructure, runs and manages offload activity through its coi_daemon.

Coprocessor offload infrastructure coprocessor offload infrastructure runs and manages offload activity through its coi_daemon.

DAPL In high-performance computing (HPC) and parallel programming, DAPL and MPI refer to the integration of the Direct Access Programming Library (DAPL) with a Message Passing Interface (MPI) library - Intel MPI. This combination is used to enable high-speed, low-latency communication between compute nodes in a cluster using Remote Direct Memory Access (RDMA) over fabrics like InfiniBand or RoCE. DAPL can also be referred to as Direct Access Provider Layer.

DDD Data Display Debugger for the MIC. GNU DDD (Data Display Debugger) is a graphical front end for the command-line debuggers GDB. DDD is a graphical user interface (using the Motif toolkit) for the command-line debugger GDB.

EMC 003-0080-01 3 Meter Infiniband 40gb QSFP to Cx4 The EMC 003-0080-01 is a Direct Attach Copper (DAC) InfiniBand/Ethernet cable of 3 meters length and 28 wire gauge 28 AWG. It has a 40Gb/s QSFP+ connector on one end and a CX4 (SFF-8470) connector on the other. It has a 40Gb/s QSFP+ (Quad Small Form-factor Pluggable) connector on one end and a CX4 (SFF-8470) connector on the other.

EMU The EMU is a specialized hardware component built directly into each core's Vector Processing Unit (VPU). It is a dedicated piece that has transcendental instructions that accelerate complex mathematical functions. The EMU allows the processor to execute complex math operations like reciprocals, square roots (sqrt), and logarithms (log) in a vectorized fashion.

EUI EUI (End User Interface) and EUI-H were the messaging and collective communication libraries developed by IBM for its early Scalable POWERparallel Systems. These libraries were optimized for the Vulcan switch interconnecting the parallel processors on the SP1 and early SP2 supercomputers.

FMA A Fused Multiply-Add (FMA) is a single processor instruction that computes the operation with only one final rounding step instead of rounding after both the multiplication and the addition. The floating-point fused multiply-add (FMA) operation was first introduced in 1990 on the IBM RS/6000 workstation using the POWER1 processor.

HCA A Host Channel Adapter (HCA) is a dedicated hardware interface card used specifically in InfiniBand architectures to connect a computer or server (the host) to an InfiniBand network fabric, switches, or target storage devices. Ultra-Low Latency & High Throughput: Designed for high-performance computing (HPC), artificial intelligence (AI/ML) clusters, and enterprise data centers, modern HCAs (such as the NVIDIA ConnectX InfiniBand Adapters) deliver massive speeds ranging from 10 Gbps up to 400 Gbps (NDR) and beyond.

HPS A High Performance Switch (HPS) network switch is a networking hardware device designed to handle massive data throughput, ultra-low latency, and high-density traffic routing between your HPMCs.

IB Commands InfiniBand (IB) commands are used to monitor, diagnose, and configure InfiniBand fabrics, host channel adapters (HCAs), and managed switches.

IBM Vulcan Switch The IBM Vulcan switch chip is an 8 × 8 crossbar switching element designed by IBM Research that powered the High-Performance Switch (HPS) in IBM's RISC System/6000 SP1 and SP2 scalable parallel computer systems. The HPS released in 1993 for SP1 RS/6000 workstations each with max 16-MB RAM, and a max of 128 compute nodes. The HPS-2 (High-Performance Switch 2) released in 1994 for SP2 RS/6000 workstations each with max 128-MB RAM, and and max of 512 compute nodes.

IBM Vulcan switch The 1994's IBM Vulcan switch chip is an 8 × 8 crossbar switching unit designed by IBM Research that powered the High-Performance Switch (HPS) in IBM's RISC System/6000 SP1 and SP2 scalable parallel computer systems. SP1/SP2 nodes participate in high-performance parallel processing by connecting to the switch via communication adapters. The SPs are distributed workstation nodes, each with its own CPU and memory, they do not share memory. Admittedly an application can run in parallel on the distributed nodes, however there was bo global memory shared by the nodes, and the SP switch was only used for message passing. Having redundant nodes attached to the swicth, this Bidirectional Multistage Interconnection Network (BMIN) was rather used for high availability of the application. The lack of shared memory also implied the lack of native Fetch&Add.

SAMPLE BOOK PREVIEW

IMCI IMIC, Intel® Initial Many Core Instructions, is a 512-bit SIMD (Single Instruction, Multiple Data) vector instruction set developed for Intel's early Many Integrated Core (MIC) architecture, such as the Knights Corner generation of Xeon Phi coprocessors.

SAMPLE BOOK PREVIEW

Intel Qlogic CBL2-1001001-3 The Intel Qlogic CBL2-1001001-3 is a 10-meter (32.81 ft) 40GbE / QDR InfiniBand Active Optical Cable (AOC) with QSFP+ to QSFP+ plug connectors. This cable has a wavelength of 850nm and a data rate 40Gbps Ethernet or QDR (Quad Data Rate) InfiniBand. It is used to interconnect the HPMCs via switches. This cable is used to connect the HPMCs via the High Performance Switch (HPS) Network Switch.

iomem The file /proc/iomem shows the current map of a Linux system's physical memory for RAM and hardware devices. It is a virtual file that does not take up disk space.

MIC Intel Many Integrated Cores, a processor architecture designed for massively parallel computing and high-performance workloads

MPI The Message Passing Interface (MPI) is a standardized, portable Application Programming Interface (API) designed for programming parallel computers and distributed-memory systems. MPI provides a library of functions and subroutines callable from C, C++, and Fortran. The MPI program runs as multiple independent processes, each with its own local memory address space, and where each process gets a unique integer identifier called a rank, starting from 0 up to N - 1 (where N is the total number of processes). The processes can communicate with each other via communicator objects.

MPI on the Intel® Xeon Phi When the Intel® MPI Library is used, the Message Passing Interface (MPI) on the Intel® Xeon Phi™, one can select the execution in either native or hybrid parallel models. The MPI ranks run on the primary Intel® Xeon® host CPU which uses offload directives (or OpenMP/threads) to push specific compute kernels onto the Xeon Phi coprocessor card. The message passing via MPI can be intermixed by using Hybrid MPI+OpenMP to explicitly combine inter-node or inter-card messages via MPI with intra-card multi-threading through OpenMP, hence saturating the high core counts (60+ cores and 240 threads) on the HPMC MIC.

MPSS The Linux mpss service is the control daemon for the Intel Manycore Platform Software Stack (Intel MPSS), which manages Intel Xeon Phi coprocessor hardware cards in high-performance computing environments. The service is run by the lightweight Linux-based micro-operating system (uOS) on each installed Xeon Phi card.

ofa The OpenFabrics Enterprise Distribution (OFED) stack for Linux used in this book was released by OpenFabrics Alliance (OFA), hence the "ofa" is often seen as: DAPL provider ofa-v2-mlx4_0-1u. The OpenFabrics Alliance (OFA) is an open-source industry consortium that develops, tests, and distributes software for high-performance, low-latency network fabrics. Founded in 2004 as the OpenIB Alliance, and renamed the

OpenFabrics Alliance in 2006. It delivered the unified cross-platform RDMA (Remote Direct Memory Access) and kernel bypass software to achieve wire-speed messaging with minimal CPU overhead. It supported InfiniBand, RoCE (RDMA over Converged Ethernet), iWARP, and OpenFabrics Interfaces (Libfabric). It released the OpenFabrics Enterprise Distribution (OFED) stack for Linux.

OFED OFED, OpenFabrics Enterprise Distribution, is an open-source Linux software stack designed for RDMA (Remote Direct Memory Access) and kernel bypass applications.

port_lid In InfiniBand network switching, the port_lid refers to the specific physical or logical port utilized by the software Subnet Manager (SM) which is the "brain" of the entire fabric. InfiniBand switching features a centralized control plane (unlike Ethernet's distributed learning), where the Subnet Manager acts as the network's "brain" to discover the topology, assign Local Identifiers (LIDs), and build switch routing tables.

Qlogic 12200-18 18-Port QDR InfiniBand Network Switch The QLogic 12200-18 is an 18-port Quad Data Rate (QDR) InfiniBand edge switch designed for high-performance computing (HPC) clusters. It is switching capacity is 2.88 Tbps aggregate bandwidth with switching latencies under 140 nanoseconds. Its 18 QSFP+ optical ports support QDR 40 Gbps, 20 Gbps, and 10 Gbps auto-negotiation. It is managed via 10/100 Ethernet RJ45 or serial RS-232 ports.

Qlogic CBL1-0600230 The Qlogic CBL1-0600230 is a 2-meter passive Direct Attach Copper (DAC) cable featuring QSFP-to-QSFP connectors designed for 40 Gbps high-speed networking. The cable has a wire gauge 30 AWG and ending with two QSFP to QSFP metal connectors. It is a Passive Direct Attach Copper (DAC) twinax cable of length 2 meters wih data rate 40 Gbps (40GBase-CR4).

Qlogic CBL1-0600230 2M Cable Cop 30 AWG STD QSFP to QSFP The QLogic CBL1-0600230 is a 2-meter passive copper Direct Attach Cable (DAC) featuring QSFP-to-QSFP connectors and 30 AWG wiring, designed for high-speed 40 Gbps network connectivity. This cable is used to connect the HPMCs via the High Performance Switch (HPS) Network Switch.

SAMPLE BOOK PREVIEW

Qlogic CBL2-1001001-3 The Qlogic CBL2-1001001-3 is a 10-meter (32.81-foot) 40Gbps QSFP+ active optical fiber network cable designed for InfiniBand connections. The cable is a 10 meters Active Optical Cable (AOC) with QSFP+ to QSFP+ (38-position plug-to-plug) connectors, and data rate up to 40Gbps (QDR InfiniBand).

QLogic InfiniPath QLogic InfiniPath is a high-performance InfiniBand host channel adapters (HCAs) and software designed specifically for High-Performance Computing (HPC) clusters.

QLogic InfiniPath QLE7140 The QLogic InfiniPath QLE7140 is a single-port, Single Data Rate (SDR) InfiniBand Host Channel Adapter (HCA) with a PCI Express x8 interface and a raw data rate of 10 Gbps (effective data rate of 8 Gbps). The HPMC is equipped with this HCA to interconnect with other HPMCs via the InfiniBand network switch.

QLogic QLE7340 The QLogic QLE7340 is a single-port 40 Gbps Quad Data Rate (QDR) InfiniBand host channel adapter (HCA) using a PCI Express Gen2 x8 interface. This HCA is a single Quad Small Form-factor Pluggable (QSFP) slot supporting 3 meters copper cables or 30 meters fiber optic cables. The adapter host interface is PCI Express Gen2 x8, it has unidirectional throughput of 3,400 MBps with ultra-low point-to-point latency under 1.0 µs. It has a message rate of roughly 30 million messages per second. It supports QSFP copper cables (0.5m to 3m) and optical cables (3m to 30m).

QSPF A QSFP (Quad Small Form-factor Pluggable) cable is a high-density, high-speed networking cable used to connect your HPMCs to HPS (High Performance Switch) network switch. The three types QSFP+, QSFP28, and QSFP56, use 4-lanes, and handle 40 Gbps, 100 Gbps, and 200 Gbps, aggregate speeds respectively. The QSFP-DD with double-density 8-lanes can handle 400 Gpbs.

QUDA QUDA (Quantum Chromodynamics on NVIDIA CUDA) is an open-source library used to accelerate lattice Quantum Chromodynamics (LQCD) calculations on graphics processing units (GPUs).

RDMA Remote Direct Memory Access (RDMA) is a networking technology that lets computers exchange data directly between their application memories over a network without involving the operating system or CPU.

RoCE RoCE, RDMA over Converged Ethernet, is a network protocol that allows computers to transfer data directly between their memories over standard Ethernet without involving the main processor (CPU) or operating system. Instead of a dedicated InfiniBand hardware fabric, it runs on standard Ethernet infrastructure.

Shared Memory Communications SMC-R Shared Memory Communications over RDMA using RoCE.

shm shm means shared memory transport (used for intra-node communication only). Set I_MPI_FABRICS=shm:dapl to have MPI startup(): shm and dapl data transfer modes.

sm_lid sm_lid (Subnet Manager Local Identifier) is the 16-bit address assigned to the active InfiniBand Subnet Manager (SM) on the fabric. In an InfiniBand network, the "brain" of the entire fabric is a software component called the Subnet Manager (SM). On the HPMC, you can read the LID directly from the kernel interface at /sys/class/infiniband//ports//sm_lid

Theano Theano is an open-source Python library focusing on deep learning frameworks. It is an optimizing compiler developed in 2007 at the Université de Montréal. Theano uses symbolic computation graphs and automatic differentiation where one defines mathematical expressions as an abstract blueprint so the Theano compiler optimizes the graphs by fusing operations and unrolling loops before compiling it into highly efficient machine code. TensorFlow and PyTorch were inspired by the Theano technical architecture.

Torch Torch was written in 2001 by the Idiap Research Institute in Switzerland. It was developed as a modular, open-source machine learning library and scientific computing framework used to build and train deep learning models. It was written in C++ and CUDA. In 2010, researchers at New York University (NYU) re-architected Torch7 in Lua (LuaJIT) to support academic deep learning research. Later on, Torch transitioned into the wildly popular Python-based framework, PyTorch.

vmallocinfo /proc/vmallocinfo is a pseudo-file in Linux that provides detailed information about virtual memory allocations made via vmalloc or ioremap.

VPU The Vector Processing Unit (VPU) in Intel Xeon Phi processors is a dedicated hardware engine inside each core that handles massive data-parallel calculations using a 512-bit Single Instruction, Multiple Data (SIMD) architecture. Each VPU processes 512 bits of data per cycle, allowing it to execute operations on multiple data elements simultaneously. The VPU includes dedicated 16-bit mask registers to enable conditional execution on specific elements inside a vector register without branching.

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

SAMPLE BOOK PREVIEW

Index

#!/usr, 46, 81, 86, 110
512-bit, 40, 71, 73, 75-77, 79, 82, 90-91, 97, 205, 217
Accelerated lattice Quantum Chromodynamics, 142
ALU, (Arithmetic Logic Unit), 71, 90, 103, 214
AVX, (Advanced Vector Extensions), 74, 205, 209, 220. *See also* AVX-512F .
AVX-512F, 74
AxBd.cu, 120-123
Benchmark, 40, 65, 67, 69, 120, 135, 143-144
BMIN, (Bidirectional Multistage Interconnection Network), 212. *See also* Vulcan .
Bootstrap, 22, 24-25, 31, 86, 182
br0, 23-26, 29, 32, 160
bridge br0, 26, 29, 32
broadcast, 79, 81-82, 88-89, 97, 102-103, 160, 164, 211
CAffE, (Convolutional Architecture for Fast Feature Embedding.), 109, 118, 131, 135-137, 209
Casting, 87, 94, 96, 99, 104
Casting Vectorized Data Type, 94, 104
  vecmul512mixCAST.cpp, 94, 96
  vecmul512mixCV.cpp, 95-96, 99
  _mm512_cvtpslo_pd, 95-96
CNN, 135
COI, (coprocessor offload infrastructure), 55. *See also* coi_daemon.
coi_daemon, 49, 53, 55-56, 173, 176-177, 179. *Same as* COI(coprocessor offload infrastructure)*See also* COI.*Same as* coprocessor offload infrastructure.*See also* COI.
Context Switching, 41, 173, 188-191, 212, 220
Conway's Game, 116
coprocessor offload infrastructure, (COI), 55. *See also* coi_daemon; COI.
cpio, 25, 29-34, 207
cpp, 41, 43, 75-76, 80, 84-87, 89-96, 99-101, 104, 135, 137, 186
cpuinfo, 40, 72-74, 84-85, 220
crossbar, 212, 214-215
CUDA, 3-4, 41, 107, 109-111, 113, 115-116, 118-119, 127-129, 131, 142, 145-150, 183, 207, 209, 219
cuda-gdb, 118-119, 219
CUDA_DEBUGGER_SOFTWARE_PREEMPTION, 119
CUDNN, 131, 183
CUDNN_PATH, 131, 183
DAPL, (Direct Access Programming Library)(Direct Access Provider Layer), 164, 170-172, 174-179, 188, 191. *See also* I_MPI_FABRICS; shm:dapl; ofa; RDMA; InfiniBand; RoCE.
dcpus, 56, 191, 196-198, 220
DDD,, 45-46, 87, 89, 91. *See also* gdb; xphiddd; Ddd*rshCommand.
Ddd*rshCommand, 45. *See also* DDD.
default_roce_mode, 164
diffusion_omp.c, 47, 49-50
dint, 41, 56, 60, 139, 188-189, 193-194, 201-203
DYLD_LIBRARY_PATH, 110, 183
EMU, (Extended Math Unit), 97-98
FIBer, 15-16
FMA, (Fused Multiply-Add), 40, 44, 65, 67, 120-121, 123, 150, 200, 217-218. *See also* fused.
FORTRAN, 75, 127-129, 211
Fused, 40-42, 73, 81, 99-100, 121-125, 150, 216-218. *See also* FMA.
GDB, 45-46, 71, 73, 75-76, 80-82, 84-86, 89, 100-101, 118-119, 121, 123, 219. *See also* DDD.
gdbmic-disas, 75, 80-81, 89, 100-101
gdbmic-reg, 75-76
gdbmic-run, 75, 84
gdbmic-step, 75, 84-86

SAMPLE BOOK PREVIEW

-ggdb, 45
GIGAFLOP, 41-44, 65-68, 75-76, 78-81, 83-86, 99-104, 120-121, 123-124, 184-187
GIGAFLOPS, 41-44, 65-68, 75-76, 78-81, 83-86, 99-104, 120-121, 123-124, 184-185
gpdata3, 200
GPU, 3-4, 8, 12, 17, 21, 41, 73, 107-110, 115-116, 118, 120, 123, 125, 127-128, 131, 133, 135-136, 138, 142, 145-150, 200, 217, 219-220
GTX TITAN X, 4, 12, 108, 115, 131, 143, 145-148, 150, 209
HCA, (Host Channel Adapter), 3-4, 13, 15, 17-18, 153-155, 177, 219. *See also* RDMA.
High Performance Switch, 211
HPS, (High Performance Switch), 211-212
IB, 4, 12, 27, 52, 76, 84, 86, 109-110, 135, 153, 155, 163-168, 170, 183, 204, 207-209, 211-217
IB Commands, (InfiniBand IB Commands), . *See also* ib0; qib0; LID.
- **ib_cm**, 156
- **ib_core**, 156
- **ib_ipath**, 155
- **ib_ipoib**, 156
- **ib_iser**, 156
- **ib_qib**, 155-156, 201
- **ib_qib0**, 201
- **ib_ucm**, 156
- **ib_umad**, 156
- **ib_uverbs**, 156
- **ib_write_bw**, 165-166
- **ib_write_lat**, 167, 221
- **ibnetdiscover**, 157
- **ibnodes**, 156
- **ibping**, 165
- **ibv_devices**, 157
- **ibv_devinfo**, 157-158, 164. *See also* port_lid.
- **ibv_rc_pingpong**, 165

IB/RoCE, 164-165
ib0, 160, 164. *See also* IB Commands.
ibacm, 154
IBM Blue Gene, 214-215
icc, 27-29, 32, 34, 38, 45, 49, 52, 55, 67-68, 180, 182
ifconfig, 24, 26, 160, 164
IMCI, (Initial Many-Core Instruction set), 74. *See also* MIC.
InfiniBand, 3-4, 7, 10, 16, 153-160, 163-168, 170, 174, 212-215, 219. *See also* QLogic QLE7340.
InfiniPath, 153, 157, 159
interrupts, 40-41, 60, 71, 73, 188-190, 193-196, 198, 201-203, 214
Intrinsic, 71, 87, 91-97, 219
Intrinsic Vector Operations, 91-93
- **vector512.cpp**, 87, 89-90
- **adding512.cpp**, 91
- **sum512.cpp**, 92
- **vecmul512f.cpp**, 93
- **vecmul512d.cpp**, 93-94

IO edge interrupts, 73
iomem,, 22-23, 108. *See also* /proc/iomem.
ioremap, 23, 108. *See also* vmallocinfo.
iperf3, 167-169
iperf3 Over Ethernet, 168
iperf3 Over Infiniband, 167
IPoIB Switching, 170
iterations, 66-69, 121-125, 135, 143-144, 146-147, 149, 165-167, 212
L1 cache, 72, 216
L2 cache, 72, 216, 220
LATENCY, 23, 73, 97-98, 108, 163, 166-167, 211-215, 218
LD_LIBRARY_PATH, 52, 76, 84, 86, 109-110, 135, 183, 207
libcudnn.so, 131, 183
LID, 157, 165-167. *See also* IB Commands.

SAMPLE BOOK PREVIEW

logical processors, 72-73, 84, 97
lspci, 22-23, 108, 155
lua, 131, 138, 183, 207, 216. *See also* torch.
MADD, 97
matrix, 54, 65-69, 120-125, 128-129, 139, 143, 145-146, 148, 150, 200, 206, 212, 214-216, 218
Matrix Multiplication, 54, 65, 67, 120, 123, 129, 200
memory mapped, 8, 22-23, 72, 220
Message Passing, 13, 154, 170, 211-213
MIC, (Intel Many Integrated Core), 4, 8-11, 17, 21-34, 36-38, 40-43, 45, 47, 49-50, 52-56, 60, 65-68, 71-81, 83-87, 89-97, 100-104, 125, 135, 154, 158-160, 170-177, 179-182, 184-186, 188-191, 193-194, 196-198, 200, 202-203, 209, 217, 219-220. *See also* micinfo; MPSS.
mic-gigaflopsDouble.cpp, 101
mic-gigaflopsDouble_xphi_Gg.MIC-ASM, 101
mic-gigaflopsFloat.cpp, 100
mic-gigaflopsFloat_xphi_Gg.MIC-ASM, 101
mic_virtblk, 27
micctrl, 27-29, 32, 34, 38, 180, 182. *See also* MPSS.
micctrl --updateramfs, 34
micinfo, 22, 28, 182. *See also* MPSS.
micnativeloadex, 50, 52
Micro Channel Architecture, 211
mkswap, 27
modprobe, 22, 27, 33-34
MPI, (Message Passing Interface), 10, 13, 21, 36, 41, 125, 142, 154-158, 165, 170-179, 184, 188, 191, 208-209, 211-214, 219. *See also* offloading; openMP.
MPI ENVIRONMENT VARIABLES,
I_MPI_DAPL_PROVIDER_LIST, 171-172, 174-176, 178-179, 188, 191
I_MPI_PERHOST, 171-172, 174-175, 178, 188, 191
I_MPI_DEBUG, 171-172, 174-176, 178-179, 188
I_MPI_FABRICS, 171-172, 174-176, 178-179, 188, 191. *See also* DAPL.
I_MPI_FALLBACK, 171-172, 174-176, 178-179, 188, 191
mpiexec.hydra, 171-172, 174-175, 178, 188, 191
MPIRUN, 170, 175
MPSS, (Manycore Platform Software Stack), 7, 22, 25-26, 28, 30-32, 34, 180-182. *See also* micctrl; MIC; micinfo.
service mpss start, 27
service mpss stop, 27
systemctl start mpss, 28, 34, 182
Native Loading, 47, 50, 193, 196
neural network, 132, 135, 206-207
NFS, 29, 34, 36, 38
nn package, 132
nvcc, 116, 118, 121-125, 128-130, 138, 209
NVIDIA, 3, 108-110, 113, 119, 125-126, 142, 145, 205, 207, 209
nvidia-settings.desktop, 113
nvvp, 125-126, 183
objdump, 75, 184-185
objdump-mic, 75, 184-185
ofa, (Open Fabrics Alliance), 171-172, 174-176, 178-179, 188, 191. *See also* DAPL; RDMA.
OFED, (OpenFabrics Enterprise Distribution), 154, 212. *See also* RDMA; RoCE; InfiniBand.*See related* RoCE.
Offloading, 21, 23, 36, 50, 54, 170-171, 174, 193, 196. *See also* MPI.
OMP, 3-5, 7-8, 11, 21-23, 26-31, 33-34, 36-38, 42-43, 45, 47-50, 52-56, 65-68, 71, 75-76, 79-83, 87, 90-91, 96-100, 102, 107-111, 115-116, 118, 121-125, 127-133, 135, 137-138, 140, 142, 145-150, 153-154, 158-159, 170-172, 174-175, 178-180, 182-184, 186, 188, 191, 193-194, 196-198, 200-202, 204-205, 207-209, 211-219
OMP_NUM_THREADS, 43, 171-172, 174-175, 178, 188, 191
openMP, 54. *See also* MPI.
PIPELINE, 71, 73, 75, 90, 96-97, 103, 217-218
U and V pipelines, 75

SAMPLE BOOK PREVIEW

SP pipeline, 71, 90, 97
DP pipeline, 71, 90, 97
ALU pipeline, 103
Scatter/gather pipeline, 97

port_lid, (InfiniBand Subnet Manager Port), 158-159. *See also* ibv_devinfo; LID.
Profiler, 125-126
Profiling, 125
python3, 133, 207-210

PyTorch, 131-132, 207. *See also* torch.

qib0, 157, 164-167, 171-172, 175-176, 179, 188, 201, 221. *See also* IB Commands.
QLOGIC, 16, 153-155, 157, 212, 214-215
QLOGIC HARDWARE,
Qlogic CBL1-0600230 2M Cable Cop 30 AWG STD QSFP to QSFP, 153
Intel Qlogic CBL2-1001001-3, 153
QLogic InfiniPath QLE7140, 153

QLogic QLE7340,, 153. *See also* InfiniBand.
EMC 003-0080-01 3 Meter Infiniband 40gb QSFP to Cx4, 153
EMC 003-0080-01 3 METER 40gb QSFP to Cx4 Cable, 153
QUDA, (Quantum Chromodynamics on NVIDIA CUDA), 142-143, 145-150
ramfs, 25-27, 32, 34
Random FOG, 116

ranks, 170-171, 173, 191, 211-213. *See related* MPI.
RDMA, (Remote Direct Memory Access), 4, 13, 153-154, 157, 163-168, 170, 174, 188, 212-215, 219. *See also* OFED; HCA card.*See related* RoCE.
readelf, 184
registers, 21, 71, 73, 75-79, 82, 87, 89-90, 93-95, 97, 103, 115, 150, 206, 211, 217-218
Rescheduling interrupts, 41, 189-190, 193-195, 201-203
RoCE, (RDMA over Converged Ethernet), 163-165, 212
Scatter, 55, 75, 79, 97, 212
Scatter/Gather, 75, 97
Shared Libraries, 36, 184

shm, (shared memory transport), 171-172, 174-176, 178-179, 188, 191. *See also* I_MPI_FABRICS; shm:dapl; RDMA.

shm:dapl, 171-172, 174-176, 178-179, 188, 191. *See also* DAPL.
shuffles, 96, 102-103
shuffling, 101-102, 104
SIMD, 21, 44, 73, 75, 219

sm_lid, (Subnet Manager Local Identifier), 158-159. *See also* LID; ibv_devinfo; port_lid.
Superscalar, 72, 75, 85, 87, 97
Superscalar Vector Processor, 75, 85, 87
Swap Space, 26
switch, 4, 13, 15-16, 23, 41, 110, 153-157, 164, 170, 173-174, 177-178, 188-191, 205, 211-215, 220
swizzle, 79, 81-82, 87-93, 98, 102-103
systemctl status opensm, 164
TENsOr, 131-133, 138-139, 205, 209
tensors, 132-133
Theano, (named after Theano of Croton), 109, 118, 131, 138-140
TLB shootdowns, 41, 189-190, 193-195, 201-203

torch,, 109-110, 118, 131-133, 183, 207-209. *See also* PyTorch; lua.
transcendental, 75, 97-98
trap_offload.c, 170
vblendmps, 103
vcvtps2pd, 96, 102-104
vcvtps2ps, 102
VECTOR, 40, 42, 71, 73-77, 79-83, 85, 87, 89-97, 99-104, 127-128, 139, 143, 145-146, 148, 150, 186, 204-205, 217-219
vector instruction, 71, 74-75, 79, 82-83, 87, 93-95
VECTOR REGISTER, 71, 73, 75-77, 79, 87, 89-90, 93-95, 97, 103
vector zmm, 81-83, 89, 102-103

vmallocinfo,, 23, 108. *See also* /proc/vmallocinfo; vmalloc; ioremap.
vmulpd, 96, 104

SAMPLE BOOK PREVIEW
SAMPLE BOOK PREVIEW

vpermf32x4, 102-103
VPU, (Vector Processing Unit), 40, 71, 73-75, 77-79, 82, 90, 97
VPU instruction, 97
Vulcan, 211-212, 215
xphiddd, 45-46. *See also* DDD.

SAMPLE BOOK PREVIEW
SAMPLE BOOK PREVIEW