Hasitha Muthumala Waidyasooriya
Masanori Hariyama
Kunio Uchiyama

Design of FPGA-Based Computing Systems with OpenCL

Springer

Design of FPGA-Based Computing Systems with OpenCL

Hasitha Muthumala Waidyasooriya
Masanori Hariyama • Kunio Uchiyama

Design of FPGA-Based Computing Systems with OpenCL

Springer

Hasitha Muthumala Waidyasooriya
Graduate School of Information Sciences
Tohoku University
Sendai, Japan

Masanori Hariyama
Graduate School of Information Sciences
Tohoku University
Sendai, Japan

Kunio Uchiyama
Hitachi (Japan)
Tokyo, Japan

ISBN 978-3-319-68160-3 ISBN 978-3-319-68161-0 (eBook)
https://doi.org/10.1007/978-3-319-68161-0

Library of Congress Control Number: 2017953437

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Heterogeneous computing uses several types of processor and accelerator to optimally execute a computational task. It is a computing architecture for a wide range of apparatus from low-power embedded applications to high-performance computing. Various accelerators are used in heterogeneous computing, and recently, field-programmable gate arrays (FPGAs) have gained a lot of attention. Because of the benefits of semiconductor miniaturization, an FPGA can now incorporate various hardware resources such as logic blocks, digital signal processor (DSP) blocks, memory blocks, and central processing unit (CPU) cores. FPGA accelerators are able to achieve excellent overall performance and performance/watt in various applications.

The traditional FPGA design methodology using hardware description language (HDL) is inefficient because it is becoming difficult for designers to develop details of circuits and control states for large and complex FPGAs. Design methodologies using C language are more efficient than HDL-based ones, but it is still difficult to design a total computing system. The C-based design tools can generate only the data path inside FPGAs and do not support interface between FPGAs and external memory devices or interface between FPGAs and host CPUs. Designers still need to use HDL to design interface circuits.

To solve these problems, a design environment based on Open Computing Language (OpenCL) has recently been brought to FPGA design. OpenCL is a C-based design environment for a heterogeneous computing platform consisting of a host CPU and accelerators, such as GPUs and FPGAs. OpenCL allows designers to develop whole computation: computation on the host, data transfer between the host and accelerators, and computation on accelerators. Hence, by analyzing OpenCL codes, the OpenCL design environment for FPGAs can generate FPGA circuits together with interface circuits.

OpenCL has many more features for FPGAs than for other accelerators because FPGAs have more flexibility. To fully take advantage of the potential performance of FPGAs using OpenCL, designers must generate appropriate FPGA-suitable codes

that will be translated into the target architecture. In this book, we systematically explain how to develop FPGA-suitable OpenCL codes and present some OpenCL design examples of FPGA accelerators.

Sendai, Japan — Hasitha Muthumala Waidyasooriya
Sendai, Japan — Masanori Hariyama
Tokyo, Japan — Kunio Uchiyama

Contents

Chapter 1
Background

1.1 Trends of Heterogeneous Computing

Heterogeneous computing uses several types of processor to optimally execute a computational task. In a heterogeneous computing system, different kinds of processor, such as a central processing unit (CPU), a digital signal processor (DSP), a media processor, a graphics processing unit (GPU), and a vector processor, execute a program in harmony to achieve high performance and reduce power consumption. This architecture has existed for over 40 years. In the field of supercomputers, vector supercomputers incorporating a CPU with a vector arithmetic processor were first commercialized in the latter half of the 1970s [1, 2]. In consumer electronics, such heterogeneous computing is used for a wide range of devices, such as video game consoles and smart phones. For example, an LSI for a video game console developed in 2005 had embedded a CPU core and eight single instruction multiple data (SIMD) processor cores to execute game software that needed to calculate a huge amount of three-dimensional graphics data in real time [3]. The system-on-a-chip (SoC) for smart phones, developed in 2016, incorporates four CPU cores, a GPU core, and a DSP core to flexibly support various mobile applications [4].

The stream of heterogeneous computing is also spreading to cloud computing. Microsoft has been using field-programmable gate arrays (FPGAs) to speed up search engines and machine learning for cloud services, thereby reducing power consumption [5]. Google has been using GPUs to accelerate machine learning [6] and has also developed the tensor processing unit (TPU) dedicated for machine learning acceleration [7]. In addition, Amazon's cloud service provides heterogeneous computing platforms in which GPUs and FPGAs are used as accelerators [8].

Recently, GPUs and FPGAs have attracted a lot of attention as accelerators of heterogeneous computing. The number of computing units that can be integrated on a chip is increasing for both of them, because of the benefits of semiconductor miniaturization. As a result, the computing performance is dramatically improved. As of 2017, the single-precision floating-point performance of GPUs and FPGAs

H.M. Waidyasooriya et al., *Design of FPGA-Based Computing Systems with OpenCL*,
https://doi.org/10.1007/978-3-319-68161-0_1

has reached 10 tera floating-point operations per second (TFLOPS), far exceeding the computational performance of CPUs. Thanks to their high computing capability, both GPUs and FPGAs have been used as accelerators in a wide range of fields, including image/signal processing, cryptographic processing, bioinformatics, CAD/CAE/CAM applications, computational fluid dynamics, financial engineering, and machine learning.

A GPU is a processor-type accelerator that mounts many arithmetic cores onto a chip. Although a GPU software designer needs to develop a parallelized program that runs on the embedded arithmetic cores in parallel in an SIMD manner, software development environments such as CUDA [9] and Open Computing Language (OpenCL) [10] have been developed for such parallel programming and make it easier for software engineers to develop programs for a GPU. On the other hand, because an FPGA can freely configure its internal logic, an FPGA accelerator can adopt any kind of architecture, such as data flow, systolic array, and SIMD/multiple instruction multiple data (MIMD) processors. By choosing the optimal architecture for algorithms or processing, an FPGA accelerator can achieve higher overall performance and performance/watt than a GPU [11–13]. However, when implementing an FPGA accelerator, it is necessary to first construct the target architecture for algorithms with hardware on the FPGA. This hardware design process has sometimes been a factor that keeps software engineers away from FPGA accelerators.

1.2 OpenCL Models for Heterogeneous Computing

The OpenCL specification is defined in four parts, which it refers to as "models." These models are described briefly in this section.

1.2.1 Platform Model

An OpenCL platform consists of one host and one or more devices as shown in Fig. 1.1. It provides an abstract hardware model for devices. A device consists of multiple compute units and a compute unit consists of multiple processing elements (PEs).

There could be several OpenCL platforms in one system. Usually, multiple platforms do not interact with each other. A platform corresponds to a vendor provided SDK (software development kit). Table 1.1 shows some of the platforms available and their corresponding SDKs. We can only use the devices that are supported by the SDK. The devices may also be restricted by their device drivers. Usually, the device driver is different for different devices. How to find the OpenCL platform and devices are explained in Sect. 2.2.3.

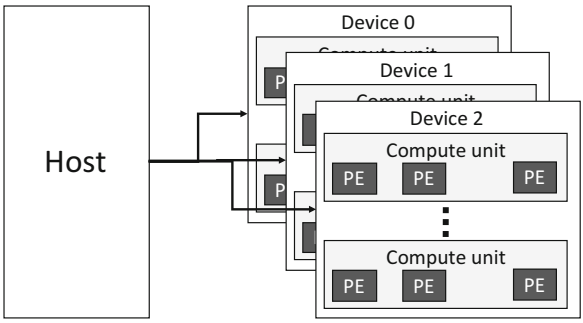


Fig. 1.1 OpenCL platform model

Table 1.1 OpenCL platforms

OpenCL platform	SDK version
CPU (host) and CPU (device)	Intel SDK for OpenCL applications
CPU (host) and GPU (device)	Nvidia SDK for OpenCL
CPU (host) and GPU (device)	AMD APP SDK 3.0 for 64-bit Linux
CPU (host) and FPGA (device)	Intel FPGA SDK for OpenCL

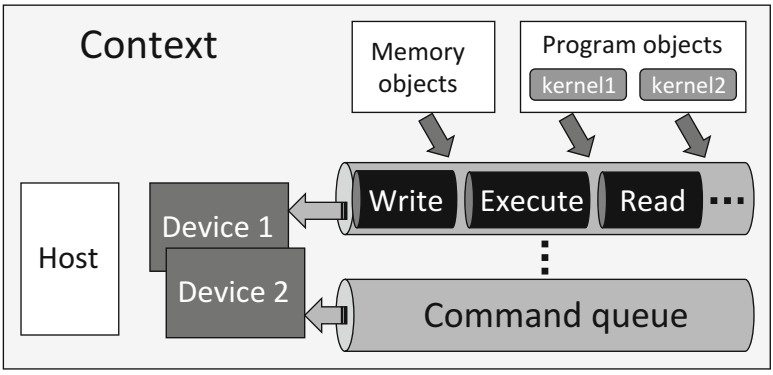


Fig. 1.2 OpenCL execution model

1.2.2 Execution Model

The execution model defines how the host communicate with devices and how and when the host execute programs on devices. Figure 1.2 shows the execution model. An OpenCL context is created with one or more devices. It provides an environment for host–device interaction, memory management, device control, etc.

A command queue is the communication mechanism that the host uses to request action of a device. At least one command queue per device must be created. The host issue commands such as data write, data read, execute kernels, etc. Those commands are stored in the command queue and issued appropriately. The host also manages the command execution order and their dependencies.

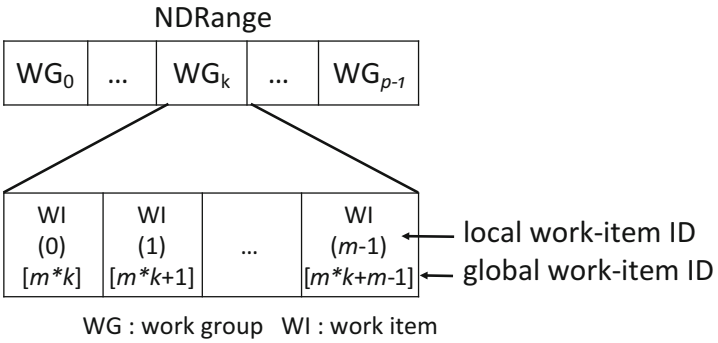


Fig. 1.3 An example of an NDRange

1.2.3 Kernel Programming Model

Kernels are functions executed on an OpenCL device. The unit of the execution of a kernel is called a work-item. Work-items are organized into work-groups. The entire collection of work-items is called an NDRange, where work-groups and work-items can be divided into N dimensions, where the maximum value of N is three.

In order to explain the relationship between the work-groups and work-items, we consider the example shown in Fig. 1.3. It shows an NDRange of one dimension. It contains p work-groups and each work-group contains m work-items. The sizes of the NDRange and work-groups are specified by the host program. Inside a kernel program, a work-item is identified by its global work-item ID and local work-item ID. The local work-item ID is used to identify a work-item inside a work-group. Therefore, two work-items belonging to different work-groups can have the same local work-item ID. The global work-item ID is used to identify any work-item in the NDRange. Therefore, each work-item has a unique global work-item ID.

1.2.4 Memory Model

Figure 1.4 shows the OpenCL memory model. The memories in the host and the device are called host and device memories, respectively. The host memory is accessible only to the host. In order for the device to use the data in the host memory, those data should first be transferred to the global memory of the device. The global memory is accessible to both the host and the device. The constant memory is a read-only memory for the device. The local memory belongs to a particular work-group. Data in the local memory of a particular work-group are shared by its work-items, whereas those data are not accessible to other work-groups. The private memory belongs to a work-item. Each work-item can have its own private memory and it is not accessible to the other work-items. The memory model is explained in detail in Chap. 5.

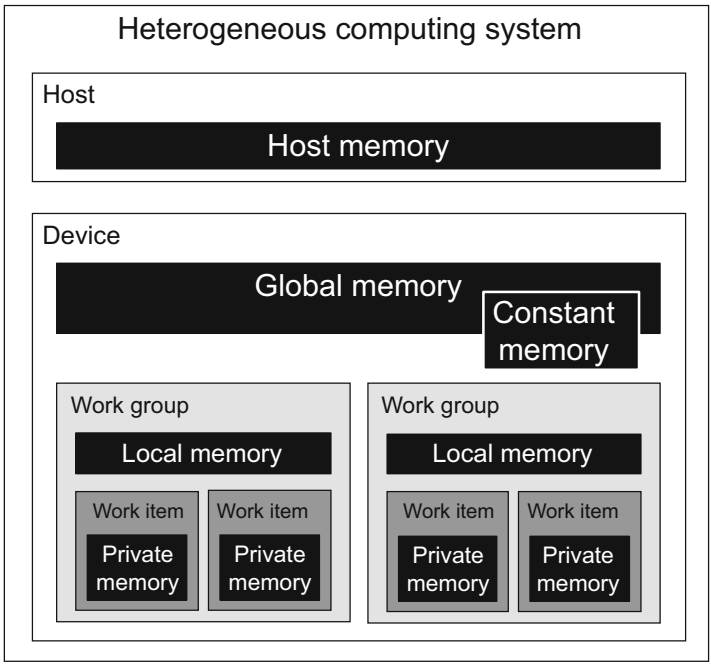


Fig. 1.4 OpenCL memory model

1.3 OpenCL for FPGA-Based Computing Systems

As described in Sect. 1.1, FPGAs are now used for developing custom accelerators for various domains, from low-power embedded applications to high-performance computing. Because of the benefits of semiconductor miniaturization technology, FPGAs have been increasing in size, containing various hardware resources such as logic blocks, DSP blocks, memory blocks, and CPU cores. Moreover, in order to develop large-scale systems, FPGAs are connected to CPUs by high-speed interfaces, such as Peripheral Component Interconnect Express (PCIe), as accelerators, and/or are connected to each other to form a cluster for high-performance computing and big data processing.

Traditional design methodologies using hardware description language (HDL) have become inefficient because it is more difficult for designers to develop details of circuits and control states for such large and complex FPGA-based systems. Design methodologies using C languages are more efficient than HDL-based ones because they allow a designer to define high-level behavioral description. Even if designers use C-based design tools, designing a total system is not very easy. C-based design tools can generate only the data path inside an FPGA; they do not support the interface between FPGAs and external memory devices, or the interface between FPGAs and host CPUs. Designers still need to use HDL to design the interface circuits.

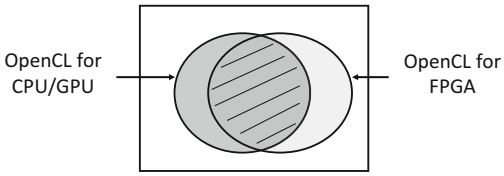


Fig. 1.5 Relationship among different OpenCL design environments

To solve these problems, a C-based OpenCL design environment has recently been brought to FPGA design. OpenCL allows designers to describe whole computation: computation on the host, data transfer between the host and accelerators, and computation on accelerators. Hence, by analyzing OpenCL codes, the OpenCL design environment for FPGAs can generate FPGA circuits together with the interface circuits.

1.4 Differences Among OpenCL-Based Design Environments

Figure 1.5 shows the relationship among different OpenCL design environments. OpenCL for FPGA is based on OpenCL 1.0, but contains many extended features to fully exploit the FPGA's potential. Therefore, the techniques and codes in this book usually do not work in other OpenCL environments [14].

References

1. IPSJ Computer Musium, FACOM 230-75 APU, http://museum.ipsj.or.jp/en/computer/super/0003.html
2. IPSJ Computer Musium, HITACH M-180 IAP, http://museum.ipsj.or.jp/en/computer/super/0004.html
3. D. Pham, S. Asano, M. Bolliger et al., The design and implementation of a first-generation CELL processor, in *ISSCC, Digest of Technical Papers* (2005), pp. 184–185
4. Qualcomm Snapdragon 821 (2017), https://www.qualcomm.com/products/snapdragon/processors/821
5. K. Ovtcharov, O. Ruwase, J.Y. Kim, J. Fowers, K. Strauss, E. Chung, Toward accelerating deep learning at scale using specialized hardware in the datacenter, in *Hot Chips 27* (2015)
6. Announcing GPUs for Google Cloud Platform (2016), https://cloudplatform.googleblog.com/2016/11/announcing-GPUs-for-Google-Cloud-Platform.html
7. N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal et al., In-datacenter performance analysis of a tensor processing unit, in *ISCA 2017* (2017)
8. Linux Accelerated Computing Instances, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/accelerated-computing-instances.html
9. CUDA (2017), https://www.geforce.com/hardware/technology/cuda
10. OpenCL (2017), https://www.khronos.org/opencl/
11. B. da Silva, A. Braeken, E.H. D'Hollander, A. Touhafi, J.G. Cornelis, J. Lemeire, Comparing and combining GPU and FPGA accelerators in an image processing context, in *2013 23rd International Conference on Field programmable Logic and Applications* (2013), pp. 1–4

12. S. Asano, T. Maruyama, Y. Yamaguchi, Performance comparison of FPGA, GPU and CPU in image processing, in *2009 International Conference on Field Programmable Logic and Applications* (2009), pp. 126–131
13. H.M. Waidyasooriya, Y. Takei, S. Tatsumi, M. Hariyama, OpenCL-based FPGA-platform for stencil computation and its optimization methodology. IEEE Trans. Parallel Distrib. Syst. **28**(5), 1390–1402 (2017)
14. D.R. Kaeli, P. Mistry, D. Schaa, D.P. Zhang, *Heterogeneous Computing with OpenCL 2.0* (Morgan Kaufmann, Cambridge, 2015)

Chapter 2
Introduction to OpenCL for FPGA

2.1 FPGA

An FPGA is a programmable hardware [1, 2] that can be reconfigured after manufacturing. It contains programmable logic gates and programmable interconnects as shown in Fig. 2.1. FPGAs also contain configurable memory modules and DSPs (dedicated multipliers). We can connect these logic gates, DSPs and memories to implement any arbitrary circuit. Therefore, many processors or accelerators can be implemented on FPGA to do different computations. FPGAs are already used in many fields such as signal processing [3, 4], high-performance computing [5, 6], machine learning [7], etc.

FPGAs are a cost-effective solution compared to ASICs (application specific integrated circuits). Hardware description languages such as Verilog HDL [8] and VHDL [9] have been used to program FPGAs. In order to design a high-performance accelerator, the programmer must have a lot of knowledge and experience about hardware design. In addition, HDL-based designs require cycle-level simulations and debugging. As a result, the accelerator design time is very large. To solve these problems, OpenCL for FPGA design has been recently introduced [10, 11].

2.2 OpenCL for FPGA

2.2.1 Heterogeneous Computing System with FPGAs

Figure 2.2 shows two types of heterogeneous computing systems based on CPUs and FPGAs. Figure 2.2a shows an SoC (system-on-chip) type computing system where a CPU and an FPGA are integrated on the same package or on the same chip [12]. The data transfer between the CPU and the FPGA is done through an internal (on-chip) bus. This type of systems are usually used in low-power embedded

H.M. Waidyasooriya et al., *Design of FPGA-Based Computing Systems with OpenCL*,
https://doi.org/10.1007/978-3-319-68161-0_2

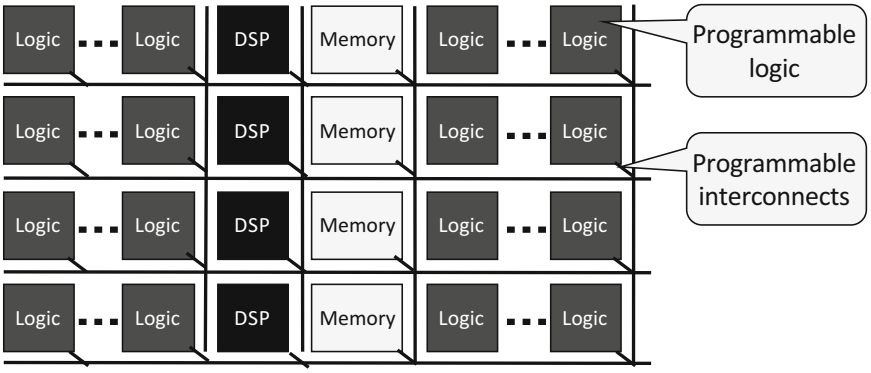


Fig. 2.1 Structure of an FPGA

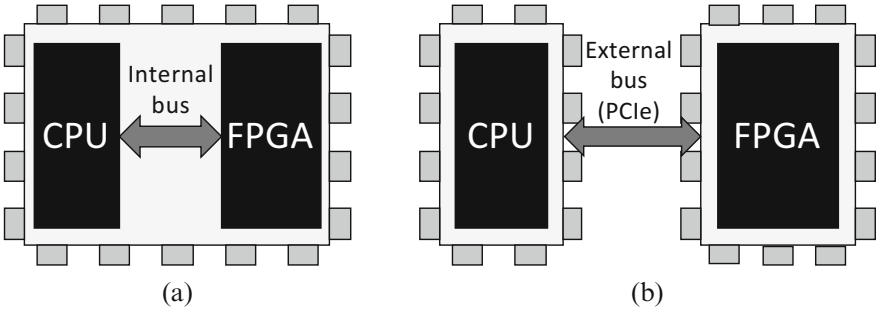


Fig. 2.2 Heterogeneous computing systems with a CPU and an FPGA. (**a**) Heterogeneous computing system used for low-power embedded processing. (**b**) Heterogeneous computing system used in high-performance computing

processing [13, 14]. Figure 2.2b shows another type of computing systems with a CPU and an FPGA. In this system, the FPGA is connected to the CPU through an external bus such as a PCI-express (PCIe). The data transfer between the CPU and the FPGA is done through the PCIe bus. This type of systems are usually used in high-performance computing [15, 16]. The scope of this book mainly covers heterogeneous computing systems with FPGAs connected through the PCIe.

There are many different FPGA boards with different input/output (I/O) resources such as external memory, PCI express, USB, etc. Moreover, there are many FPGAs that are designed for different purposes such as low-power applications and high-performance computing. The logic and memory resources of those FPGAs are significantly different. As a result, it is usually difficult to use the same code on different FPGA boards, without knowing the I/Os, logic and memory resources. Figure 2.3 shows how this problem is solved. OpenCL for FPGA uses a board support package (BSP) that contains logic and memory information, and also I/O controllers such as DDR3 controller, PCI controller, etc. as shown in Fig. 2.3. During the compilation, the kernels are merged with the BSP. As a result, the kernels can access the I/Os through the BSP. Since the BSP takes care of the I/Os and FPGA resources, the same kernel code can be used in any OpenCL-capable FPGA board.

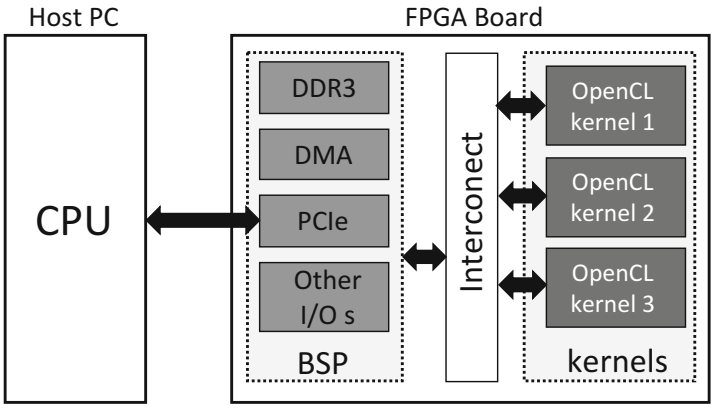


Fig. 2.3 System overview of OpenCL for FPGA

In order to use the OpenCL kernel, you have to compile it using the BSP information of the target FPGA board. When the offline-compiler version changes, some BSPs may not work with the newer version. In such cases, the board vendor may provide a new BSP version. Note that, you can write your own BSP according to the instructions given in "Intel FPGA SDK for OpenCL Custom Platform Toolkit User Guide" [17]. However, designing your own BSP is a difficult task and it requires extensive knowledge about FPGAs.

2.2.2 Setting Up the OpenCL Design Environment

This section explains how to set up the OpenCL design environment for a heterogeneous computing system with a CPU and an FPGA (Fig. 2.2b). To use OpenCL for FPGA, you need compiler software, BSP, and license files. To compile a kernel, you need a workstation with more than 24 GB of memory. We suggest to have 64 GB or more memory for smoother compilation. The software installation procedure is shown in Fig. 2.4. The details of each step are explained in the following sections.

2.2.2.1 Software Installation

The first step of the setup process is to install Quartus compiler software, SDK for OpenCL and the BSP. The current compiler is the "Quartus prime edition," and supports SDK for OpenCL of version from 15.1 to 17.0 [18]. Different versions support different FPGA types. Table 2.1 summarizes the compatibility of different versions. In this book, we use the Linux versions of all software and the operating system is CentOS 7.

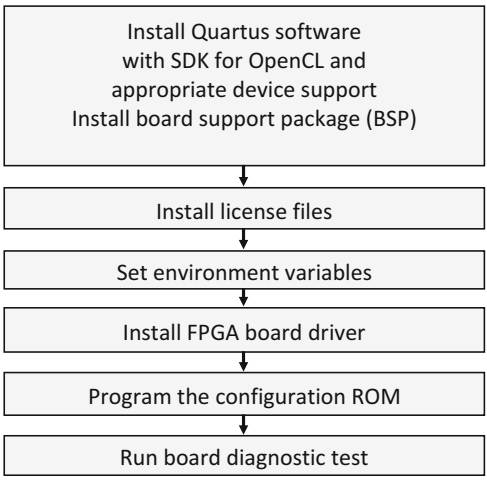


Fig. 2.4 Setup process of OpenCL design environment

Table 2.1 Compatibility of the different Quartus prime versions

Software version	License	Compatibility
Quartus prime lite	Free	Max II,V,10, Cyclone IV,V Arria II
Quartus prime standard	Standard license	Max II,V,10, Cyclone IV,V Arria II,V, Arria 10, Stratix IV-V
Quartus prime prime	Pro license (standard license included)	Arria 10

Instructions for the BSP installation are provided by the BSP vendor or the board vendor. Usually, the BSP files are copied to the `board` folder in the following path.

```
<Quartus_installation_directory>/hld/board
```

In addition, you also have to install GNU development tools such as *gcc* (include *g++*) and *make*. Those are required to build the FPGA board driver and applications in Linux environment.

2.2.2.2 License File Installation

You need a license to use Quartus prime standard or pro editions, and Intel FPGA SDK for OpenCL [19]. You may also need a license to use some of the IP cores included in BSP. There are two types of license, fixed license and floating license. Fixed license are installed on the workstation that is used for the compilation.

Floating license are installed on a server and are used by remote workstations. You have to configure the FlexLM license server as instructed in the "Intel FPGA Software Installation and Licensing" manual [20]. The number of simultaneous compilations depends on the number of slots available in the license.

The license files are referred by the environment variable `LM_LICENSE_FILE`. The value of this variable is the path or paths to the license files. You can set different paths by separating them with colons. You can also refer the paths to floating license files and fixed ones together as shown below.

```
LM_LICENSE_FILE /license/file1.dat:/license/file2.dat:
1800@server
```

2.2.2.3 Setting Environment Variables

The third step is to set the environment variables. Environment variable `QUARTUS_ROOTDIR` specifies the Quartus installed path. In CentOS, you can add the environment variables to the *.bashrc* file. If you use other linux distributions, refer the manuals to find how to set the environment variables. An example of the environment variables in CentOS is shown below.

```
export QUATUS_64BIT=1
export LM_LICENSE_FILE=1800@192.168.1.80:
  /usr/.../license/lic1512.dat
export QUARTUS_ROOTDIR=/usr/.../16.1/quartus
export QSYS_ROOTDIR="$QUARTUS_ROOTDIR"/sopc_builder
   /bin
export ALTERAOCLSDKROOT=/usr/.../16.1/hld
export AOCL_BOARD_PACKAGE_ROOT=
  "$ALTERAOCLSDKROOT"/board/terasic/de5net
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:
  "$QUARTUS_ROOTDIR"/linux64:
  "$AOCL_BOARD_PACKAGE_ROOT"/linux64/lib:
  "$ALTERAOCLSDKROOT"/host/linux64/lib
export PATH=$PATH:"$QUARTUS_ROOTDIR"/bin:
  "$ALTERAOCLSDKROOT"/linux64/bin:
  "$ALTERAOCLSDKROOT"/bin:/usr/.../16.1/qsys/bin
```

You can detect whether the environment variables of the FPGA board are set correctly by the following command.

```
aoc --list-boards
Board list:
  de5net_a7
```

If the board is detected correctly, you will get a board name such as `de5a_net_i2`. This board name is required when you program the configuration ROM and also when you compile kernels.

2.2.2.4 FPGA Board Driver Installation

If the FPGA board is detected, you can proceed to install the driver. You need administrator privilege to install the driver. The following command is used to install the driver.

```
aocl install
```

If the driver installation is successful, you can see a message similar to the one shown in Fig. 2.5.

```
[root@workstation user]# aocl install
aocl install: Running install from /usr/local/altera/16.1_pro/hld/board/
terasic/de5a_net_i2/linux64/libexec
Looking for kernel source files in /lib/modules/3.10.0-514.6.1.el7.x86_64/
build
Using kernel source files from  /lib/modules/3.10.0-514.6.1.el7.x86_64/
build
Building driver for BSP with name de5a_net_i2
make: Entering directory `/usr/src/kernels/3.10.0-514.6.1.el7.x86_64'
  CC [M]  /tmp/opencl_driver_PVc9EP/aclpci_queue.o
  CC [M]  /tmp/opencl_driver_PVc9EP/aclpci.o
  CC [M]  /tmp/opencl_driver_PVc9EP/aclpci_fileio.o
  CC [M]  /tmp/opencl_driver_PVc9EP/aclpci_dma.o
  CC [M]  /tmp/opencl_driver_PVc9EP/aclpci_pr.o
  CC [M]  /tmp/opencl_driver_PVc9EP/aclpci_cmd.o
  LD [M]  /tmp/opencl_driver_PVc9EP/aclpci_de5a_net_i2_drv.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /tmp/opencl_driver_PVc9EP/aclpci_de5a_net_i2_drv.mod.o
  LD [M]  /tmp/opencl_driver_PVc9EP/aclpci_de5a_net_i2_drv.ko
make: Leaving directory `/usr/src/kernels/3.10.0-514.6.1.el7.x86_64'
```

Fig. 2.5 FPGA board driver installation

2.2.2.5 Programming the Configuration ROM

The configuration ROM of the FPGA board should be programmed when you use a new FPGA board first time, or when you change the BSP. After the configuration ROM is programmed, restart the FPGA board to enable the new configuration. Usually, the board vendor provides configuration data to program the configuration ROM. The configuration data file has the `.aocx` extension. The configuration ROM is programmed using the following command.

```
aocl flash <board_name> <configuration_data>.aocx
```

The `<board_name>` is the name obtained using `aocl --list-boards` command. Note that, you have to configure the USB blaster in CentOS to program the configuration ROM.

2.2.2.6 Running the Diagnostic Test

After all of the above-mentioned steps are completed, your FPGA board is ready to use. You can run the "board diagnostic test" to verify the OpenCL design environment setup. In this test, the host CPU writes data to all address of the external memory. Then the host reads back and verifies the data. After that, the host calculates the write and read speeds and the memory access throughput.

In order to execute the board diagnostic test, you have to find the device names. If there are several FPGA boards in the workstation, those are identified by a unique device name. The following command gives the device names of the FPGA boards.

```
aocl diagnose
```

Figure 2.6 shows the output of the `aocl diagnose` command. As you can see, the device name is `acl0`.

You can select a device (`acl0`) and run the board diagnostic test using the following command.

```
aocl diagnose <device_name>
```

An example of the diagnostic test results is shown in Fig. 2.7. If you get the message `DIAGNOSTIC_PASSED`, the OpenCL design environment setup is successful.

2.2.3 Writing the First OpenCL Program for FPGA

In OpenCL for FPGA, you have to write two types of codes, the host code and the device (or kernel) code as shown in Fig. 2.8. The host code is used to program the CPU. The host code is written in C-language and is compiled by a C-compiler such as `gcc`. The kernel code is responsible for the computation on an FPGA. The

```
Verified that the kernel mode driver is installed on the host machine.

Using board package from vendor: Altera Corporation
Querying information for all supported devices that are installed on the
host machine ...


Device Name   Status   Information

acl0          Passed   Altera's Preferred Board
                       PCIe dev_id = AB00, bus:slot.func = 02:00.00, at
Gen 2 with 8 lanes
                       FPGA temperature = 49 degrees C.


Found 1 active device(s) installed on the host machine. To perform a full
diagnostic on a specific device, please run
      aocl diagnose <device_name>

DIAGNOSTIC_PASSED
```

Fig. 2.6 FPGA board information

kernel code is written in OpenCL which is also similar to C-language. In order to provide the FPGA bit-stream, the kernel code is compiled and merged with the BSP by the offline compiler included in the SDK for OpenCL.

2.2.3.1 Kernel Code

Let us consider our first OpenCL program that works as shown in Fig. 2.9. It adds an integer `40` to all elements in the array `k_din`, and writes the output results to the array `k_dout`. Listing 2.1 shows the OpenCL kernel code for this computation. This code is written as a single work-item kernel. There is another type of kernel called an NDRange kernel. Section 4.1 explains the kernel types in detail.

```
1 #define N (1024*1024)
2
3 __kernel void FirstProgram( __global const int * restrict k_din,
4                             __global int * restrict k_out )
5 {
6   for(unsigned i=0; i<N; i++)
7     k_dout[i] = k_din[i] + 40;
8 }
```

Listing 2.1 Kernel code for the computation shown in Fig. 2.9

2.2.3.2 Host Code

The host code performs the following tasks.

- Obtain an OpenCL platform and devices.
- Create a context and a command queue.
- Program the FPGA.

```
Verified that the kernel mode driver is installed on the host machine.

Using platform: Altera SDK for OpenCL
Using Device with name: de5net_a7 : Altera's Preferred Board
Using Device from vendor: Altera Corporation
clGetDeviceInfo CL_DEVICE_GLOBAL_MEM_SIZE = 4294967296
clGetDeviceInfo CL_DEVICE_MAX_MEM_ALLOC_SIZE = 4293918720
Memory consumed for internal use = 1048576
Actual maximum buffer size = 4293918720 bytes
Writing 4095 MB to global memory ...
Write speed: 2103.07 MB/s [2102.97 -> 2103.16]
Reading and verifying 4095 MB from global memory ...
Read speed: 3108.73 MB/s [3107.71 -> 3110.46]
Successfully wrote and readback 4095 MB buffer

Transferring 8192 KBs in 16 512 KB blocks ... 1671.24 MB/s
Transferring 8192 KBs in 8 1024 KB blocks ... 1979.42 MB/s
Transferring 8192 KBs in 4 2048 KB blocks ... 2363.58 MB/s
Transferring 8192 KBs in 2 4096 KB blocks ... 2746.89 MB/s
Transferring 8192 KBs in 1 8192 KB blocks ... 2880.71 MB/s

PCIe Gen2.0 peak speed: 500MB/s/lane

Writing 8192 KBs with block size (in bytes) below:

Block_Size Avg    Max    Min    End-End (MB/s)
  524288 1109.96 1193.22 1051.37 16.76
 1048576 1458.80 1525.09 1382.43 42.24
 2097152 1695.10 1811.86 1652.89 134.60
 4194304 1918.25 1988.36 1852.91 90.73
 8388608 2022.33 2022.33 2022.33 2022.33

Reading 8192 KBs with block size (in bytes) below:

Block_Size Avg    Max    Min    End-End (MB/s)
  524288 1536.89 1671.24 1434.46 25.11
 1048576 1891.25 1979.42 1785.03 107.15
 2097152 2298.85 2363.58 2259.39 244.76
 4194304 2673.30 2746.89 2603.56 240.50
 8388608 2880.71 2880.71 2880.71 2880.71

Write top speed = 2022.33 MB/s
Read top speed = 2880.71 MB/s
Throughput = 2451.52 MB/s

DIAGNOSTIC_PASSED
```

Fig. 2.7 Results of the diagnostic test

- Allocate memory.
- Transfer the input data from the host to the device.
- Execute the kernel.
- Transfer the output results from the device to the host.
- Release the allocated memory.

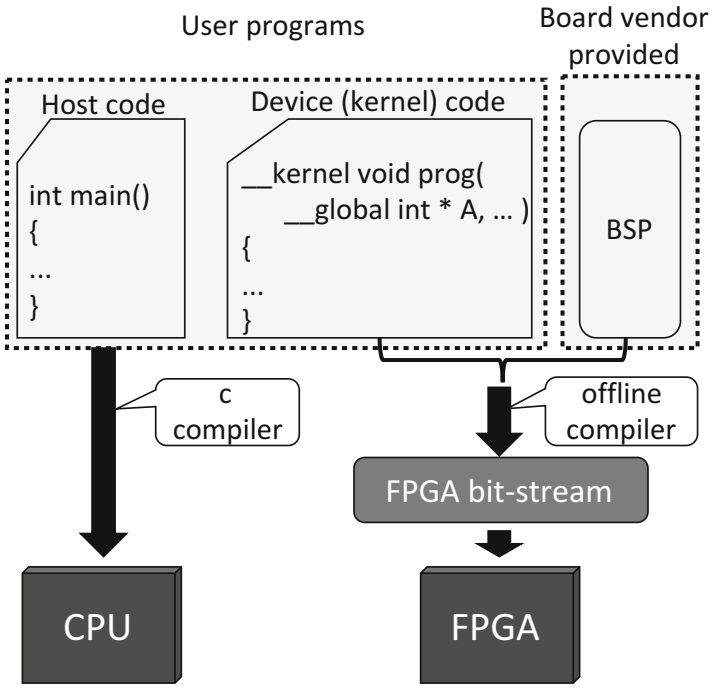


Fig. 2.8 Flow to compile host and device codes

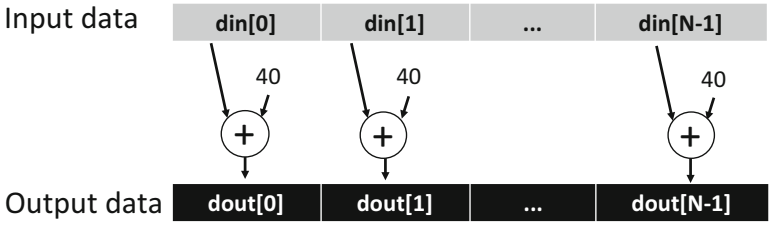


Fig. 2.9 The computation of our first OpenCL program

These tasks should be basically executed in this order. We explain the tasks below in detail.

Obtain an OpenCL Platform and Devices

This task is to obtain the platform handler of the OpenCL platform by the following code.

```
cl_platform_id fpga_paltform = NULL;
clGetPlatformIDs(1, &fpga_paltform, NULL);
```

The device is obtained as follows.

```
cl_device_id fpga_device = NULL;
clGetDeviceIDs(fpga_paltform, CL_DEVICE_TYPE_ALL, 1, &fpga_device, NULL);
```

Create a Context and a Command Queue

A context is created as follows.

```
cl_context context = clCreateContext(NULL, 1, &fpga_device, NULL, NULL,NULL);
```

Then a command queue for the context is created.

```
cl_command_queue queue = clCreateCommandQueue(context, fpga_device, 0, NULL);
```

Program the FPGA

The function `fread` in C-language is used to read the FPGA-configuration-data-file as follows.

```
size_t length = 0x10000000;
unsigned char *binary = (unsigned char *)malloc(length);
FILE *fp = fopen("FirstProgram.aocx", "rb");
fread(binary, length, 1, fp);
fclose(fp);
```

The binary file `FirstProgram.aocx` contains the FPGA-configuration-data and is obtained by compiling the kernel code using the offline compiler. Then the program is created as follows.

```
cl_program program = clCreateProgramWithBinary(context, 1, &fpga_device, &
    length, (const unsigned char **)&binary, NULL, NULL);
```

The kernel is created as follows.

```
cl_kernel kernel = clCreateKernel(program, "FirstProgram", NULL);
```

The second argument is a kernel name. As the kernel name, we set “FirstProgram” which is given in Listing 2.1.

Allocate Memory

We allocate the host memory and prepare the input data to be processed in the FPGA. The following code allocates two arrays `h_din` and `h_dout` of “ `sizeof(int)*N`” bytes, and initializes them.

```
int *host_din, *host_dout;
posix_memalign((void **)(&host_din), 64, sizeof(int)*N);
posix_memalign((void **)(&host_dout), 64, sizeof(int)*N);

for(int i=0; i<N; i++)
{
  host_din[i] = i;
  host_dout[i] = 0;
}
```

In order to access the device memory, memory objects must be created. A memory object specifies the memory location in the device. The following code creates two memory objects `mobj_din` and `mobj_dout` to store input and output data in the device, respectively.

```
cl_mem dev_din = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(int)*N, NULL,
      NULL);
cl_mem dev_dout = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(int)*N,
     NULL, NULL);
```

Memory objects `mobj_din` and `mobj_dout` correspond to host memory arrays `h_din` and `h_dout`, respectively. Hence, the size of each memory object must be equal to that of its corresponding host memory array. Note that `mobj_din` is defined as read-only and `mobj_dout` is defined as write-only. If a memory object is used for both read and write, `CL_MEM_READ_WRITE` flag should be set as the second argument of `clCreateBuffer`.

Transfer the Input Data from the Host to the Device

Before executing the kernel, the input data for the kernel is transferred from the host to the device as follows.

```
clEnqueueWriteBuffer(queue, dev_din, CL_TRUE, 0, sizeof(int)*N, host_din, 0,
     NULL, NULL);
```

Since we used `CL_TRUE` flag, `clEnqueueWriteBuffer` function returns after the data transfer is completed.

Execute the Kernel

In order to execute the kernel, the arguments of the kernel function should be specified. There are two arguments in the kernel shown in Listing 2.1. The first one is the input data `k_din` and the second one is the output data `k_dout`. The following code specifies `mobj_din` as the first argument and `mobj_dout` as the second argument.

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_din);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &dev_dout);
```

The second argument of `clSetKernelArg` specifies the argument number; values 0 and 1 represent the first and second arguments of the kernel function, respectively.

The following code executes the kernel.

```
cl_event kernel_event;
clEnqueueTask(queue, kernel, 0, NULL, &kernel_event);
clWaitForEvents(1, &kernel_event);
clReleaseEvent(kernel_event);
```

Variable `kernel_event` of `cl_event`-type is used to detect the completion of the kernel execution. Function `clWaitForEvents` waits until `kernel_event` is set.

Transfer the Output Results from the Device to the Host

The output results are stored to the device memory `k_dout` by the kernel, and the host can access this location using the memory object `mobj_dout`. The following code transfers the output results in the device to the host memory `h_dout`.

```
clEnqueueReadBuffer(queue, dev_dout, CL_TRUE, 0, sizeof(int)*N, host_dout, 0,
NULL, NULL);
```

Since we used `CL_TRUE` flag, `clEnqueueReadBuffer` function returns after the data transfer is completed.

We display a part of the computation results using the following code.

```
for(int i=N-10; i<N; i++)
    printf("din[%d] = %d; dout[%d] = %d\n", i, host_din[i], i, host_dout[i]);
```

Release the Allocated Memory

Finally, we make sure that there are no commands remained in the command queue, and then release the allocated memory in both the device and the host as follows.

```
clFlush(queue);
clFinish(queue);
//device side
clReleaseMemObject(dev_din);
clReleaseMemObject(dev_dout);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);

//host side
free(host_din);
free(host_dout);
```

Listing 2.2 shows the complete host code after putting everything together.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <CL/opencl.h>
5
6  #define N (1024*1024)
7
8  int main()
9  {
10   // Search for an openCL platform
```

```
11  cl_platform_id fpga_paltform = NULL;
12  clGetPlatformIDs(1, &fpga_paltform, NULL);
13
14  // Search for an openCL device
15  cl_device_id fpga_device = NULL;
16  clGetDeviceIDs(fpga_paltform, CL_DEVICE_TYPE_ALL, 1, &fpga_device, NULL);
17
18  // Create a context.
19  cl_context context = clCreateContext(NULL, 1, &fpga_device, NULL, NULL, NULL)
      ;
20
21  // Create a command queue.
22  cl_command_queue queue = clCreateCommandQueue(context, fpga_device, 0, NULL);
23
24  // Read FPGA binary
25  size_t length = 0x10000000;
26  unsigned char *binary = (unsigned char *)malloc(length);
27  FILE *fp = fopen("FirstProgram.aocx", "rb");
28  fread(binary, length, 1, fp);
29  fclose(fp);
30
31  // Create program
32  cl_program program = clCreateProgramWithBinary(context, 1, &fpga_device, &
      length,
33    (const unsigned char **)&binary, NULL, NULL);
34
35  // Create kernel.
36  cl_kernel kernel = clCreateKernel(program, "FirstProgram", NULL);
37
38  // Host side data
39  int *host_din, *host_dout;
40  posix_memalign((void **)(&host_din), 64, sizeof(int)*N);
41  posix_memalign((void **)(&host_dout), 64, sizeof(int)*N);
42
43  for(int i=0; i<N; i++)
44  {
45    host_din[i] = i;
46    host_dout[i] = 0;
47  }
48
49  // Create memory Object
50  cl_mem dev_din = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(int)*N,
      NULL, NULL);
51  cl_mem dev_dout = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(int)*N,
      NULL, NULL);
52
53  // FPGA side data
54  clEnqueueWriteBuffer(queue, dev_din, CL_TRUE, 0, sizeof(int)*N, host_din, 0,
      NULL, NULL);
55
56  // Execute kernel
57  clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_din);
58  clSetKernelArg(kernel, 1, sizeof(cl_mem), &dev_dout);
59  cl_event kernel_event;
60  clEnqueueTask(queue, kernel, 0, NULL, &kernel_event);
61  clWaitForEvents(1, &kernel_event);
62  clReleaseEvent(kernel_event);
63
64  // Read data from FPGA
65  clEnqueueReadBuffer(queue, dev_dout, CL_TRUE, 0,sizeof(int)*N, host_dout, 0,
      NULL, NULL);
66
67  // Print output
68  for(int i=N-10; i<N; i++)
69    printf("din[%d] = %d; dout[%d] = %d\n", i, host_din[i], i, host_dout[i]);
70
71  clFlush(queue);
```

```
72   clFinish(queue);
73   // Free memory
74   clReleaseMemObject(dev_din);
75   clReleaseMemObject(dev_dout);
76   clReleaseKernel(kernel);
77   clReleaseProgram(program);
78   clReleaseCommandQueue(queue);
79   clReleaseContext(context);
80
81   free(host_din);
82   free(host_dout);
83
84   return 0;
85 }
```

Listing 2.2 Complete host code

2.2.4 Compilation and Execution

We compile the code shown in Listing 2.1 for the Terasic "DE5a-Net Arria 10 FPGA Development Kit" [21], using the following command.

```
aoc FirstProgram.cl -o FirstProgram.aocx
 --board de5net_a7 --report
```

Option `--report` enables reporting the compilation messages as shown in Fig. 2.10. If the compilation is successfully finished, binary file `FirstProgram.aocx` is generated. This file is used to configure the FPGA by the host. Figure 2.11 shows the execution results. We can see that the computation results are accurate.

```
[user@workstation user]$ aoc FirstProgram.cl -o FirstProgram.aocx --board
de5net_a7 --report &
[2] 30869
[user@workstation user]$ aoc: Selected target board de5net_a7

+--------------------------------------------------------------------+
; Estimated Resource Usage Summary                                   ;
+----------------------------------------+---------------------------+
; Resource                               + Usage                     ;
+----------------------------------------+---------------------------+
; Logic utilization                      ;   16%                     ;
; ALUTs                                  ;   10%                     ;
; Dedicated logic registers              ;    7%                     ;
; Memory blocks                          ;   14%                     ;
; DSP blocks                             ;    0%                     ;
+----------------------------------------+---------------------------+
```

Fig. 2.10 Compilation messages for Listing 2.1

```
[user@workstation user]$ ./bin/HostProgram
Reprogramming device [0] with handle 1
din[1048566] = 1048566; dout[1048566] = 1048606
din[1048567] = 1048567; dout[1048567] = 1048607
din[1048568] = 1048568; dout[1048568] = 1048608
din[1048569] = 1048569; dout[1048569] = 1048609
din[1048570] = 1048570; dout[1048570] = 1048610
din[1048571] = 1048571; dout[1048571] = 1048611
din[1048572] = 1048572; dout[1048572] = 1048612
din[1048573] = 1048573; dout[1048573] = 1048613
din[1048574] = 1048574; dout[1048574] = 1048614
din[1048575] = 1048575; dout[1048575] = 1048615
```

Fig. 2.11 Execution results of our first OpenCL program

2.3 HDL vs. OpenCL

This section describes the advantages and disadvantage of OpenCL for FPGA over HDL.

2.3.1 Advantages

2.3.1.1 Designing with C-Language

In OpenCL-based design, we can use C-like OpenCL codes to design an FPGA accelerator. Therefore, we can dramatically reduce the design time. On the other hand, HDL-based design takes a much longer time for coding, simulation, and debugging.

2.3.1.2 Support for I/Os

OpenCL for FPGA comes with many common I/O controllers such as memory controllers, PCIe controllers, DMA controllers, etc. It also contains PCIe device drivers and an Application Programming Interface (API) to control an FPGA and to transfer data. On the other hand, HDL designers are required to design all these controllers and also the device drivers and APIs. This would take a long design time and a lot of efforts. This also requires comprehensive knowledge about how I/O controllers work and how to implement them.

2.3.1.3 Compatible and Re-usable on Different Type of FPGA Boards

OpenCL kernel codes designed for one FPGA board will also work on other type of FPGA boards. This is done by re-compiling the code using the BSP of the particular type of boards. On the other hand, an HDL code for one board may not work on a different type of boards because of the differences in the type of the external memory (DDR3, DDR4, etc.), the memory capacity, the number of memory modules, the logic resource, the DSP resource, network I/Os, etc. Therefore, a major re-design is often required, when using a different type of boards.

2.3.1.4 Easy to Debug

The functional behavior of the OpenCL codes can be emulated on a CPU. You can use `printf` in a kernel to easily collect intermediate results. You also can compile the kernel for profiling and collect information such as memory access at runtime. On the other hand, the logic-level emulation is used for HDL-based design. The logic-level emulation is extremely slower than the function-level emulation used for OpenCL-based design. Moreover, to collect intermediate results at runtime, you must design circuits for debugging inside the FPGA. It requires a lot of time and knowledge to implement the debugging circuits.

2.3.2 Disadvantages

2.3.2.1 Architecture is Hidden from the Programmer

The OpenCL programmers do not directly design the architecture. They only write the OpenCL code and the offline compiler automatically transfers it to an HDL design. Therefore, the programmers do not have a complete picture of the resulting architecture. They can only guess the resulting architecture and confirm those guesses by observing the behavior of the accelerator. Therefore, in order to obtain a better architecture, the programmers are required to know how the offline compiler translates the OpenCL codes into hardware. Otherwise, improving the performance is not easy.

2.3.2.2 Cannot Design for a Specified Clock Frequency

The clock frequency is automatically determined by the offline compiler. Therefore, the programmer cannot lower the clock frequency for the purposes such as low-power.

2.3.2.3 Difficult to Control Resource Utilization

The offline compiler usually compiles kernels for the maximum performance. Therefore, even when some of the kernels do not require the high performance, programmers cannot save resources for the purposes such as low power.

2.3.3 Improvements Expected in Future

The SDK for OpenCL version 16.1 and above allows us to use custom HDL designs with OpenCL. As a result, you can design custom functions such as 128-bit floating-point adders, that are not available in OpenCL, and integrate them with OpenCL.

IP-based design will be a new feature in OpenCL design. You can re-use the same function as an IP with its place-and-route (P&R) information. As a result, recompilation is not required for re-used IPs and this reduces the compilation time dramatically.

Although writing an OpenCL code is easy, tuning it for the optimal performance is difficult. Therefore, performance tuning and automatic optimization of OpenCL codes will be an important issue [22–24].

Since FPGAs are getting larger and larger, it is difficult for human to design an optimal code by efficiently using all FPGA resources, in a limited design time. Therefore, if we consider the design time as a constraint, the performances of the OpenCL-based designs will overtake those of the HDL-based designs. Even today, the performances of the OpenCL-based designs are not far behind those of the HDL-based designs [25].

References

1. W. Wolf, *FPGA-Based System Design* (Pearson Education, London, 2004)
2. A. Rahman, *FPGA Based Design and Applications* (Springer Publishing Company Incorporated, Berlin, 2008)
3. U. Meyer-Baese, U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, vol. 65 (Springer, Berlin, 2007)
4. A.S. Dawood, S.J. Visser, J.A. Williams, Reconfigurable FPGAS for real time image processing in space, in *International Conference on Digital Signal Processing Proceedings (DSP)*, vol. 2 (2002), pp. 845–848
5. M.C. Herbordt et al., Achieving high performance with FPGA-based computing. Computer **40**(3), 50–57 (2007)
6. P. Sundararajan, High performance computing using FPGAs, in *Xilinx White Paper: FPGAs* (2010)
7. C. Zhang, L. Peng, S. Guangyu, G. Yijin, X. Bingjun, C. Jason, Optimizing FPGA-based accelerator design for deep convolutional neural networks, in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2015), pp. 161–170
8. S. Brown, Z. Vranesic, *Fundamentals of Digital Logic Design with Verilog Design* (McGraw-Hill, New York, 2007)

9. S. Brown, *Fundamentals of Digital Logic Design with VHDL Design* (McGraw-Hill, Dubuque, 2008)
10. T.S. Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, J. Freeman, D.P. Singh, S.D. Brown, OpenCL for FPGAs: prototyping a compiler, in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)* (2012), pp. 3–12
11. T.S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, D.P. Singh, From opencl to high-performance hardware on FPGAS, in *International Conference on Field Programmable Logic and Applications (FPL)* (2012), pp. 531–534
12. Terasic, DE10 standard (2017). http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=1081&PartNo=1
13. W. Chen, Z. Wang, Q. Wu, J. Liang, Z. Chai, Implementing dense optical flow computation on a heterogeneous FPGA SoC in C. ACM Trans. Archit. Code Optim. **13**(3), 1–25 (2016)
14. S. Dhote, P. Charjan, A. Phansekar, A. Hegde, S. Joshi, J. Joshi, Using FPGA-SoC interface for low cost IoT based image processing, in *International Conference on in Advances in Computing, Communications and Informatics (ICACCI)* (2012), pp. 1963–1968
15. K. Sano, F. Kono, N. Nakasato, A. Vazhenin, S. Sedukhin, Stream computation of shallow water equation solver for FPGA-based 1D tsunami simulation. ACM SIGARCH Comput. Archit. News **43**(4), 82–87 (2016)
16. H.M. Waidyasooriya, Y. Takei, S. Tatsumi, M. Hariyama, OpenCL-based FPGA-platform for stencil computation and its optimization methodology. IEEE Trans. Parallel Distrib. Syst. **28**(5), 1390–1402 (2017)
17. Intel FPGA SDK for OpenCL Custom Platform Toolkit User Guide (2017). https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/ug_aocl_custom_platform_toolkit.pdf
18. Intel, Quartus Prime Pro Edition Download Center (2017). http://dl.altera.com/?product=qprogrammer#tabs-4
19. Intel FPGA SDK for OpenCL (2017). https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html
20. Intel FPGA Software Installation and Licensing (2017). https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/quartus_install.pdf
21. Terasic, DE5a-Net Arria 10 FPGA Development Kit (2017). https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=228&No=970&PartNo=2
22. Q. Jia, H. Zhou, Tuning stencil codes in OpenCL for FPGAs, in *IEEE 34th International Conference on Computer Design (ICCD)* (2016), pp. 249–256
23. K. Krommydas, R. Sasanka, W. Feng, Bridging the FPGA programmability-portability gap via automatic OpenCL code generation and tuning, in *IEEE 27th International Conference on Application—Specific Systems, Architectures and Processors (ASAP)* (2016), pp. 213–218
24. T. Endo, H.M. Waidyasooriya, M. Hariyama, Automatic optimization of OpenCL-based stencil codes for FPGAs, in *18th IEEE/ACIS International Conference on Software Engineeriang, Artificial Inteligence, Networking and Parallel/Distributed Computing (SNPD)* (2017)
25. K. Hill, S. Craciun, A. George, H. Lam, Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA, in *IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (IEEE, New York, 2015), pp. 189–193

Chapter 3
FPGA Accelerator Design Using OpenCL

3.1 Overview of the Design Flow Using OpenCL

Figure 3.1 shows the OpenCL-based design flow. It contains three main phases, emulation, performance tuning, and execution [1]. In the emulation phase, we verify the behavior of the code by executing it on a CPU. In the performance tuning phase, we find the performance bottlenecks by referring the compilation reports and profile information. Then we improve the performance by removing the bottlenecks. In the execution phase, we execute the OpenCL program on an FPGA-based computing system to evaluate the real performance. In the next sections, we describes these phases in detail.

3.2 Emulation Phase

In order to verify the behavior of the OpenCL kernel code, you can emulate it on a CPU. If the behavior is not observed, you have to re-write the code and emulate it again. For emulation, you have to compile the code using the following command.

```
aoc -march=emulator <kernel_code>.cl -o
 <emulate_binary>.aocx
```

The FPGA kernel code is `<kernel_code>.cl` and the compiled binary file is `<emulate_binary>.aocx`. Note that compiling for emulation can be done in a short time, and usually it takes from a few seconds to a few minutes. The host code is compiled using any C-compiler such as `gcc`. To execute the compiled binary, you have to use the following command, where the name of the compiled binary file is `<host_binary>`.

```
CL_CONTEXT_EMULATOR_DEVICE_ALTERA=1 ./<host_binary>
```


H.M. Waidyasooriya et al., *Design of FPGA-Based Computing Systems with OpenCL*, https://doi.org/10.1007/978-3-319-68161-0_3

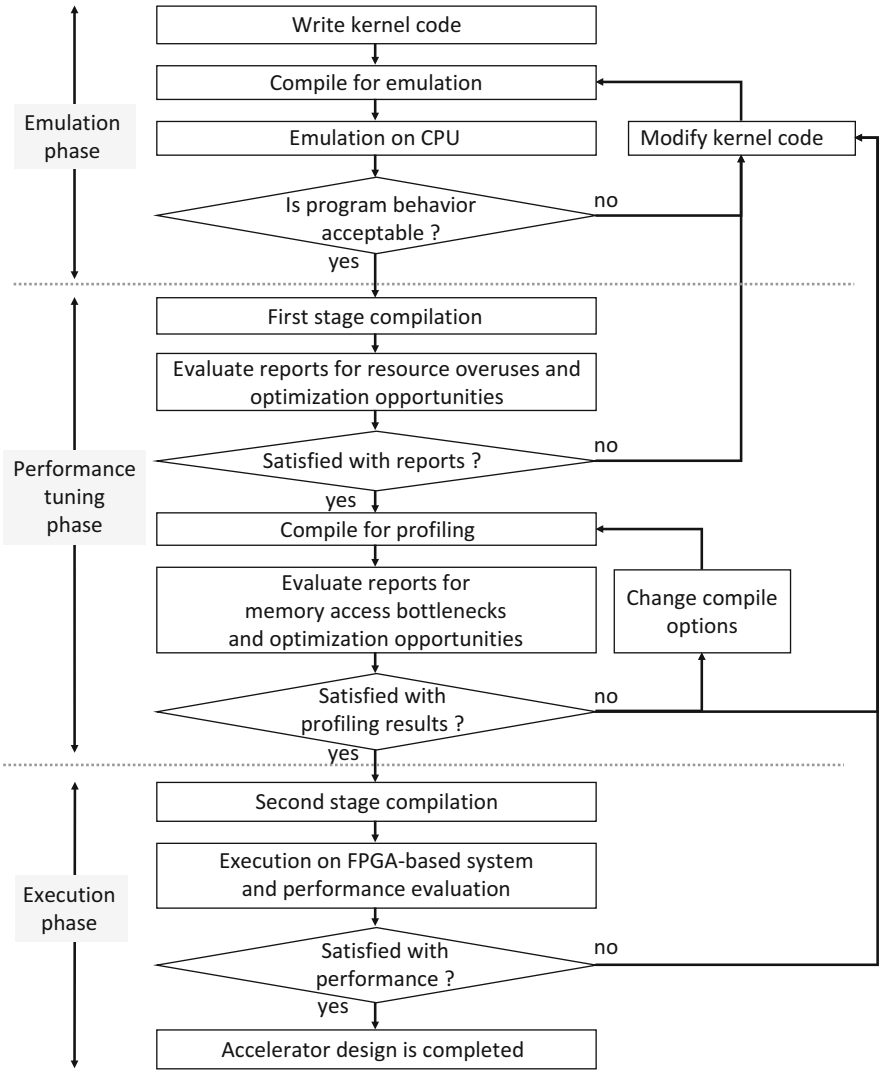


Fig. 3.1 OpenCL-based design flow

Note that you need a BSP for emulation although you do not need an FPGA board.

In emulation, the code is executed sequentially similar to a typical C-code. The parallel operations such as pipelines, loop-unrolling, and SIMD operations are ignored in emulation. Therefore, the emulation time could be very large and you may not able to emulate the whole computation. You may have to use a small data sample for emulation. Note that emulation of I/O channels is not possible until SDK for OpenCL version 16.1. However, version 17.0 provides a method to replicate the I/O channel behavior [2].

3.3 Performance Tuning Phase

In the performance tuning phase, we analyze the compilation reports to find performance bottlenecks which are caused by serial executions, large initiation intervals, etc. We can also profile the code to find information such as memory-access bandwidth at run-time. Using such information, we can modify the code for better performance [4, 5]. This section explains how to review and understand the compilation reports and profiling result.

3.3.1 Reviewing the Compilation Reports

OpenCL for FPGA uses two stage compilation process. In the first stage compilation of the performance tuning phase, the kernel code is converted into an HDL code. During the compilation, compilation reports are generated. The first stage compilation is done by the following command, where the kernel code is `<kernel_code>.cl` and the output is `<intermediate_binary>.aoco`.

```
aoc -c <kernel_code>.cl -o <intermediate_binary>.aoco
```

The compilation reports are different depending on the SDK for OpenCL versions. Until version 16.0, the compilation report is given by `<kernel_name>.log` file, where `<kernel_name>` is the kernel file name. Since version 16.1, the compilation report is given by `report.html`. Although we explain the report information using `report.html`, you can find the same information in `<kernel_name>.log` file.

Figure 3.2 shows a screen-shot of `report.html`. The **view reports pane** is shown at the top. In the **view reports pane**, you can select several types of reports such as loop analysis report, estimated resource utilization report, etc. After one of the reports is selected, it appears on the **analysis pane**. To the right of the **analysis pane**, you can see the **source code pane** where the kernel code is displayed. The **details pane** can be seen at the bottom and shows detailed report information.

3.3.1.1 Loop Analysis Report

Figure 3.3 shows a screen-shot of a *loop analysis* report. It displays loop analysis information such as pipelined information, bottlenecks, initiation interval (II), unrolling information, etc. The **details pane** at the bottom shows detailed descriptions. For example, the *loop analysis* report shows an iteration interval (II) of 7 cycles due to a data dependency bottleneck. In the **details pane**, it shows that the floating-point add operation in line 9 of the kernel code is the cause of the bottleneck. How to overcome such bottlenecks is explained in Sect. 4.4.

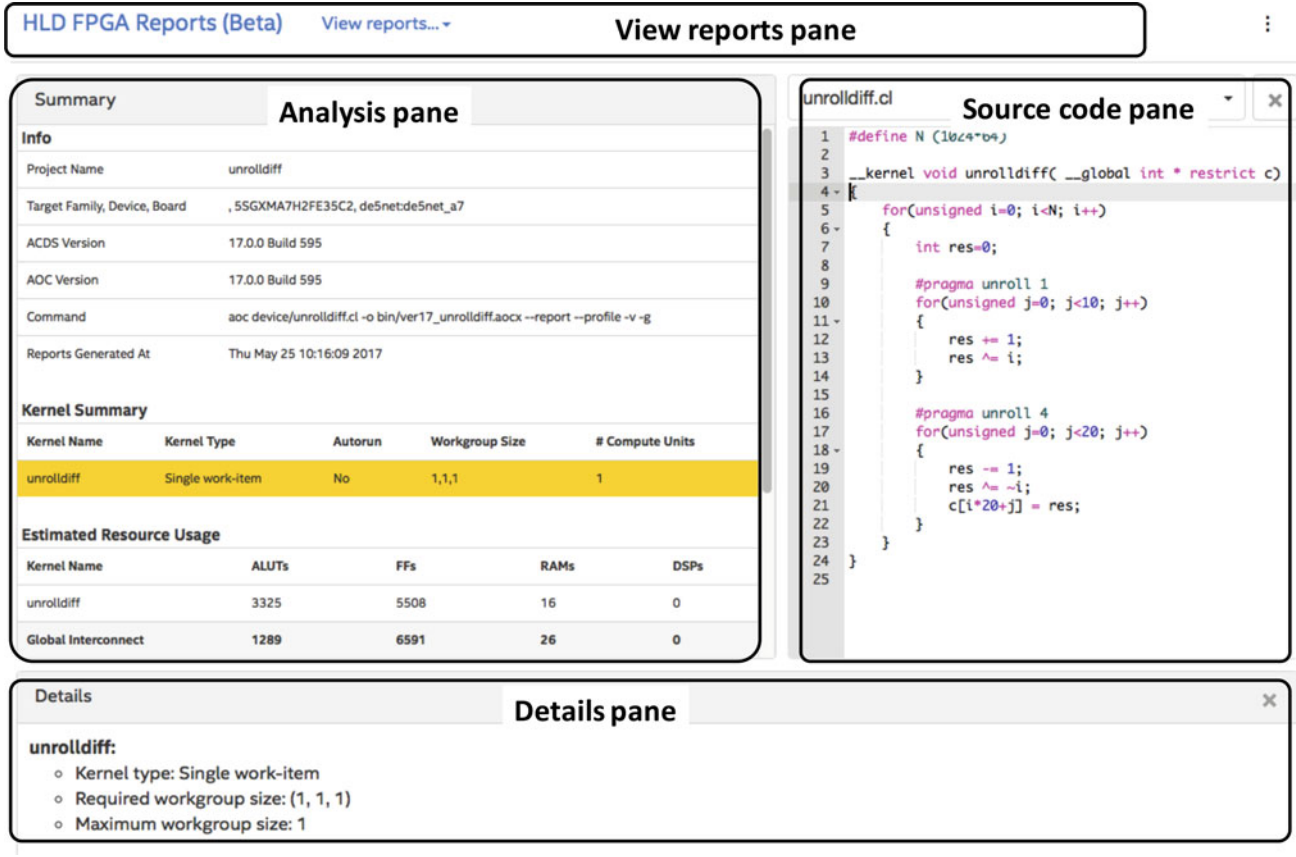


Fig. 3.2 Screen-shot of `report.html`

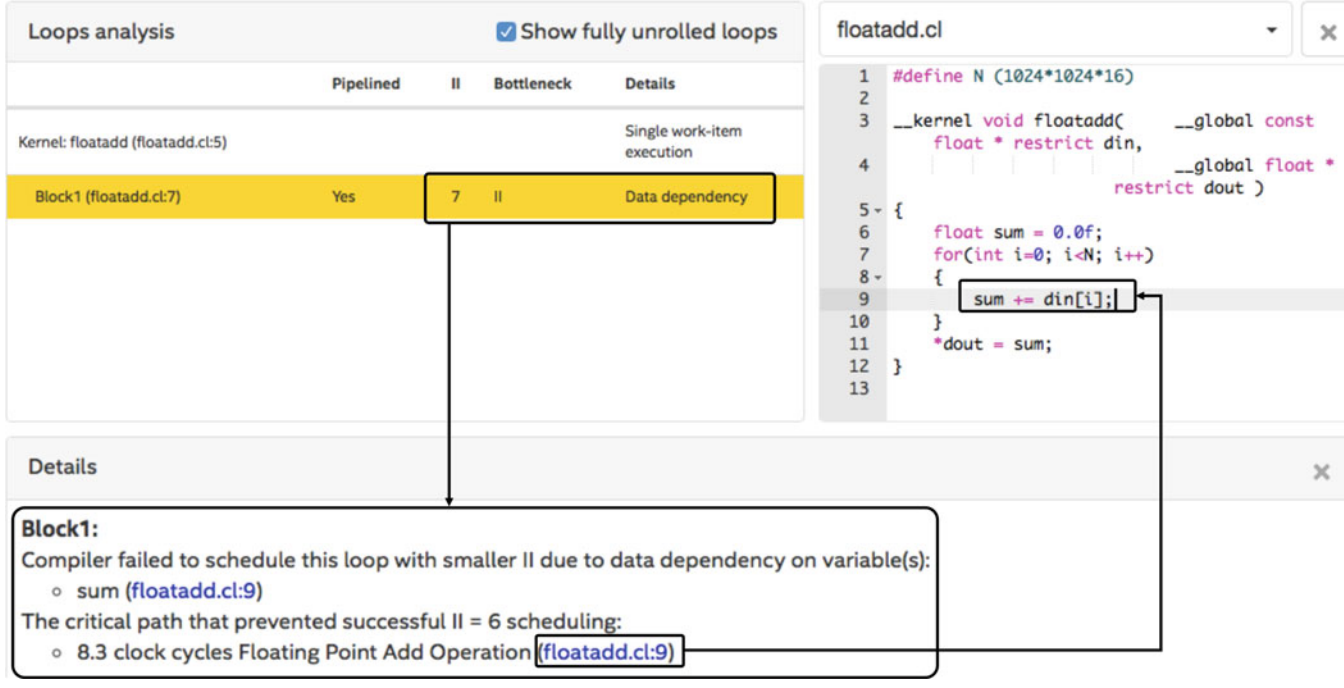


Fig. 3.3 Loop analysis report

3.3.1.2 Estimated Resource Utilization Reports

The estimated resource utilization information is available in two types of reports as shown in Fig. 3.4. One is the *area analysis of system*. An example of the *area analysis of system* is shown in Fig. 3.4a. In this report, the kernel is represented as a combination of blocks. These blocks correspond to those in the *system viewer* explained in Sect. 3.3.1.3. The resource utilization is shown for each block. Another report is *area analysis of source*, and Fig. 3.4b shows an example. This report provides resource utilization information corresponding to each kernel code line.

3.3.1.3 System Viewer

System viewer shows the structure of an FPGA accelerator. It shows a kernel as a combination of blocks. We can see how the blocks are connected to construct a kernel and how multiple kernels are connected to construct an accelerator as shown in Fig. 3.5. System viewer allows you to review information of a clicked block such as memory access width, latency and coalescing. If there is a block with red background, it indicates that the block contains performance bottlenecks.

3.3.2 Profiling

Profiling a kernel is used to analyze the behavior of the memory access. You can compile your kernel for profiling by adding `--profile` option to the compilation command. Compilation with this option generates performance counters in the FPGA to acquire the memory access information at runtime. When the kernel is executed, the profiling result is stored in `profile.mon` file. Note that, when a kernel is compiled for profiling, its clock frequency could be lower than the clock frequency of the same kernel compiled without `--profile` option. Therefore, if you want to evaluate the processing time, do not use `--profile` option.

You can evaluate the profiling result using the "dynamic profiler" software. The command to invoke the dynamic profiler depends on the SDK for OpenCL version. Use the following command for versions up to 16.1.

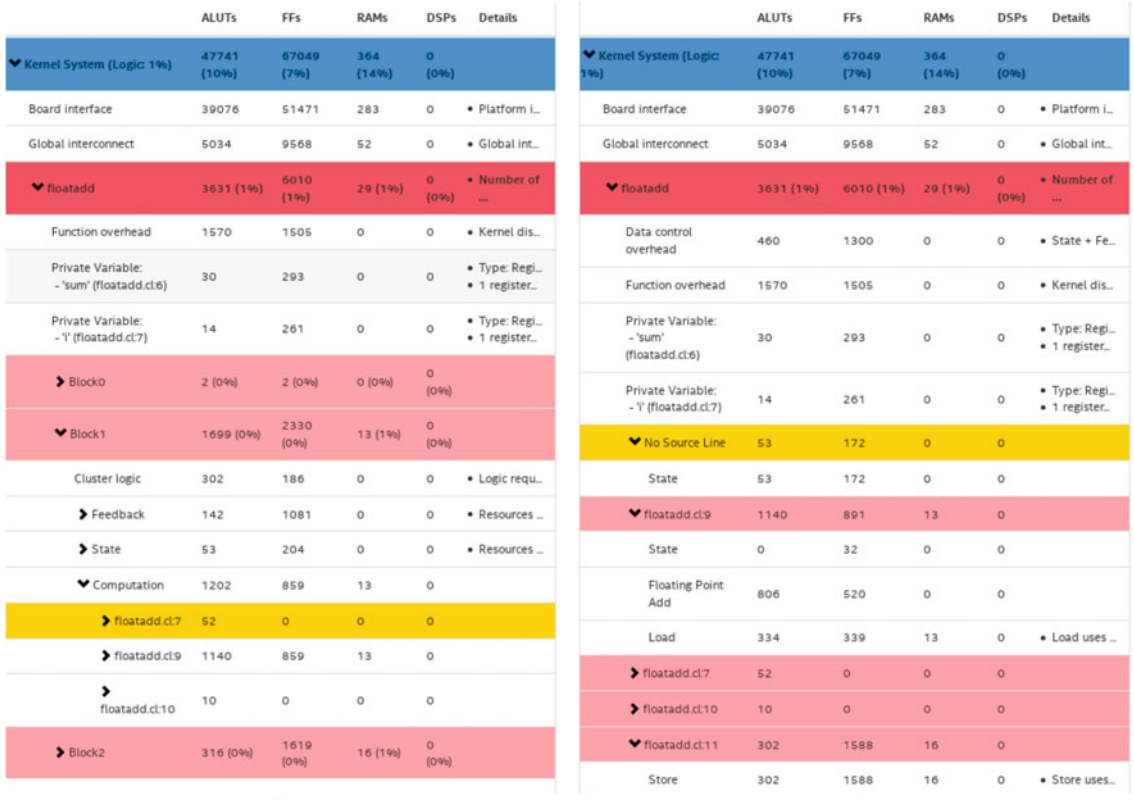
```
aocl report <compiled_binary>.aocx profile.mon
```

Use the following command for versions from 17.0.

```
aocl report <compiled_binary>.aocx profile.mon
 <kernel_file>.cl
```

The OpenCL kernel code is specified by `<kernel_file>.cl`, and the compiled binary output is `<compiled_binary>.aocx`.

Figure 3.6 shows a screen-shot of the dynamic profiler. It contains a *header*, a *source code* tab, and a *kernel execution* tab. The *header* shows the board name



	ALUTs	FFs	RAMs	DSPs	Details
Kernel System (Logic: 1%)	47741 (10%)	67049 (7%)	364 (14%)	0 (0%)	
Board interface	39076	51471	283	0	• Platform i...
Global interconnect	5034	9568	52	0	• Global int...
floatadd	3631 (1%)	6010 (1%)	29 (1%)	0 (0%)	• Number of ...
Function overhead	1570	1505	0	0	• Kernel dis...
Private Variable: - 'sum' (floatadd.cl:6)	30	293	0	0	• Type: Regi... • 1 register...
Private Variable: - 'i' (floatadd.cl:7)	14	261	0	0	• Type: Regi... • 1 register...
Block0	2 (0%)	2 (0%)	0 (0%)	0 (0%)	
Block1	1699 (0%)	2330 (0%)	13 (1%)	0 (0%)	
Cluster logic	302	186	0	0	• Logic requ...
Feedback	142	1081	0	0	• Resources ...
State	53	204	0	0	• Resources ...
Computation	1202	859	13	0	
floatadd.cl:7	52	0	0	0	
floatadd.cl:9	1140	859	13	0	
floatadd.cl:10	10	0	0	0	
Block2	316 (0%)	1619 (0%)	16 (1%)	0 (0%)	

(a)

	ALUTs	FFs	RAMs	DSPs	Details
Kernel System (Logic: 1%)	47741 (10%)	67049 (7%)	364 (14%)	0 (0%)	
Board interface	39076	51471	283	0	• Platform i...
Global interconnect	5034	9568	52	0	• Global int...
floatadd	3631 (1%)	6010 (1%)	29 (1%)	0 (0%)	• Number of ...
Data control overhead	460	1300	0	0	• State + Fe...
Function overhead	1570	1505	0	0	• Kernel dis...
Private Variable: - 'sum' (floatadd.cl:6)	30	293	0	0	• Type: Regi... • 1 register...
Private Variable: - 'i' (floatadd.cl:7)	14	261	0	0	• Type: Regi... • 1 register...
No Source Line	53	172	0	0	
State	53	172	0	0	
floatadd.cl:9	1140	891	13	0	
State	0	32	0	0	
Floating Point Add	806	520	0	0	
Load	334	339	13	0	• Load uses ...
floatadd.cl:7	52	0	0	0	
floatadd.cl:10	10	0	0	0	
floatadd.cl:11	302	1588	16	0	
Store	302	1588	16	0	• Store uses...

(b)

Fig. 3.4 Area analysis reports. (**a**) Area analysis of system. (**b**) Area analysis of source

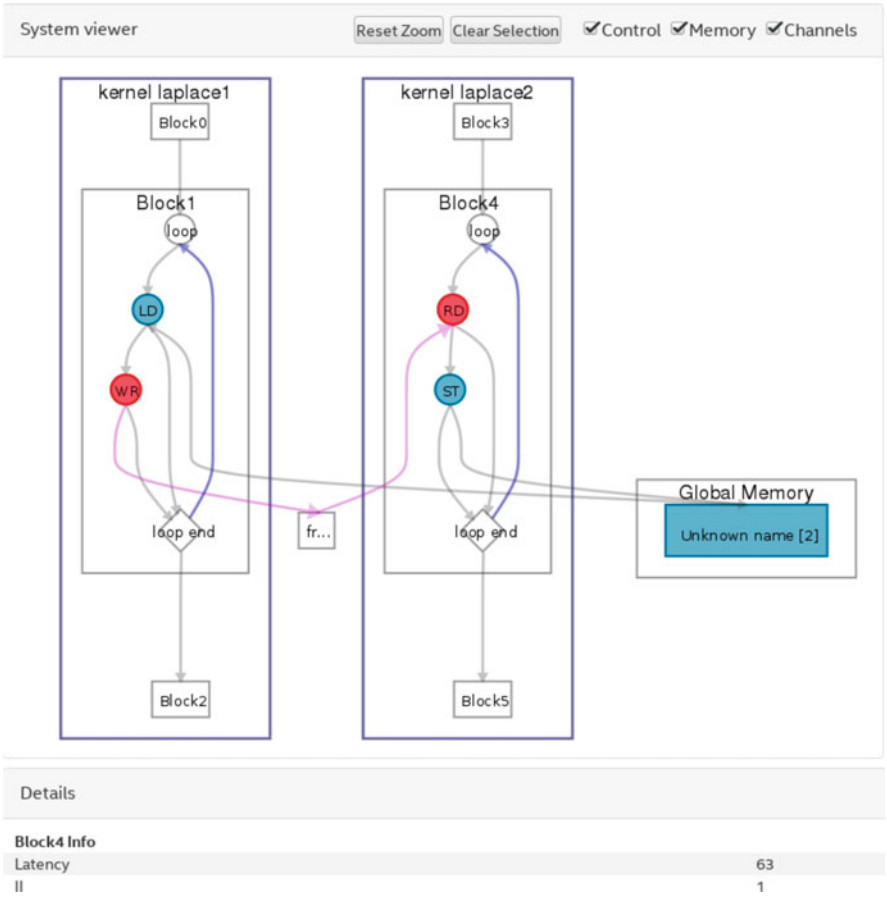


Fig. 3.5 System viewer

and the global memory bandwidth. The global memory bandwidth is the maximum theoretical bandwidth available for each memory type. The *source code* tab shows the profiling result of memory and channel accesses. The *kernel execution* tab shows the time chart of the kernel executions and the data transfers. Figure 3.6, the *source code* tab is selected.

3.3.2.1 Source Code Tab

The profiling result shown in the *source code* tab consists of four columns: attribute column, stall percentage column, occupancy percentage column, and bandwidth column.

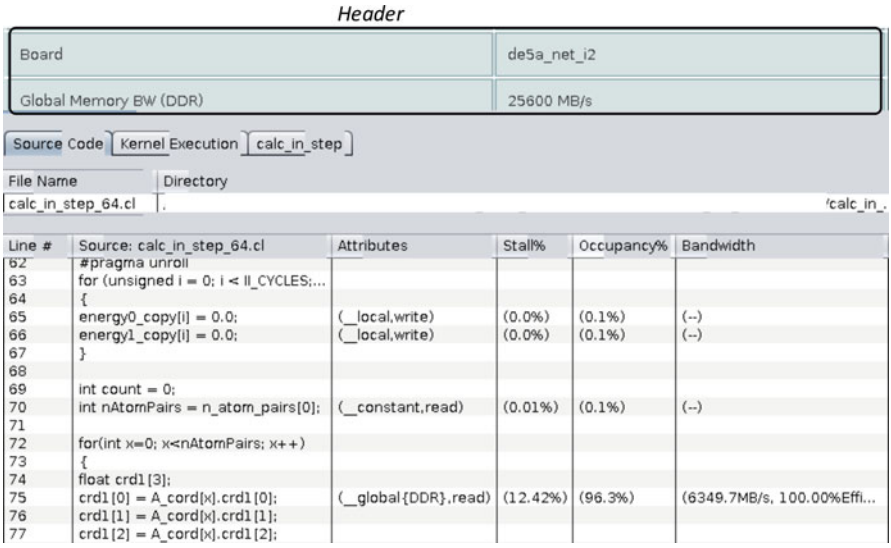


Fig. 3.6 Screen-shot of the dynamic profiler

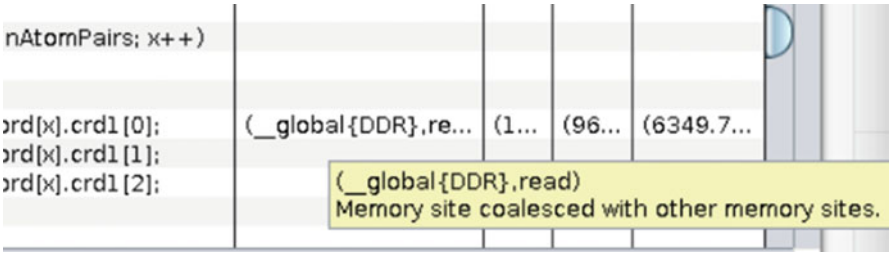


Fig. 3.7 Tool-tip text

Attributes Column

This column shows the memory type (local, global, etc.), and access type (read or write). If there are more than one memory or channel operations in one source code line, those information appears in a drop-down list-box. When you move the mouse pointer over those information, tool-tip will appear as shown in Fig. 3.7.

You can see the following information in tool-tip texts.

Cache hit percentage	This shows the percentage of global-memory access using cache. A high cache-hit rate reduces the memory bandwidth utilization.
Unaligned access	This shows the percentage of unaligned global memory access. Unaligned access is inefficient.

Coalesced access — This shows whether the load or store operations are coalesced. Memory coalescing merges multiple memory accesses of consecutive addresses into a single access.

Stall Percentage Column

This is the percentage of time that a pipeline is stalled due to memory or channel access.

Occupancy Percentage Column

Occupancy refers to the percentage of the time required for memory (or channel) access in total processing time. If the memory or channel operation is in a critical path of the kernel code and the occupancy is low, it implies that a performance bottleneck exists because work-items or loop iterations are not being issued in the hardware. Low occupancy could also appear when there is a small number of loop-iterations or a small number of work-items. In these situations, the pipeline is not fully filled in most of the kernel execution time. How to interpret the occupancy information is explained in Sect. 3.3.3.

When you move the mouse pointer over the occupancy value, a tool-tip called **activity** can appear. **Activity** is the percentage of time a predicated channel or memory instruction is enabled (that is, when conditional execution is true). Therefore, it is possible to have a high occupancy with low activity.

Bandwidth Column

Bandwidth column shows the global-memory access throughput during the kernel execution. That is the total accessed bytes during kernel execution time. The overall efficiency is the percentage of total-used-bytes of the total-accessed-bytes. Efficiency is shown in the bandwidth column in Fig. 3.6. The tool-tip shows the average **burst size** of the memory operation.

3.3.2.2 Kernel Execution Tab

Figure 3.8 shows a screen-shot of a *kernel execution* tab. It shows the time chart of executions of the kernels and data transfers between the host and the device. Note that, you have to set the environment variable ACL_PROFILE_TIMER to 1 to get the data transfer information. The other tabs denoted by the kernel names show the summary of that particular kernel execution.

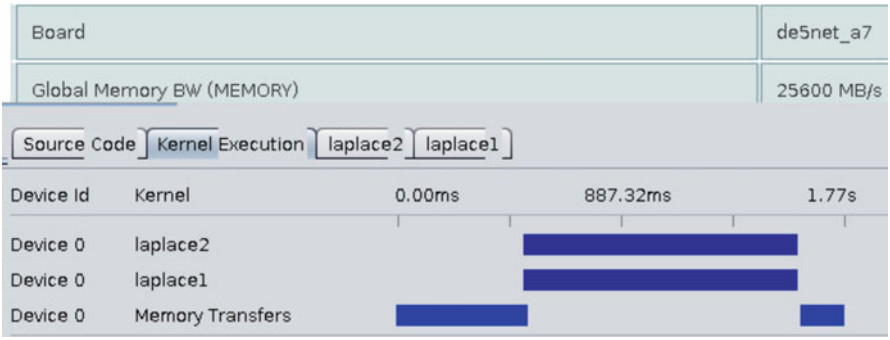


Fig. 3.8 The kernel execution tab

3.3.3 Interpreting the Profiling Result

In this section, we explain how to interpret the profiling results and how to improve performance based on that. The ideal situation of a kernel is to have 0% stalls, 100% occupancy, and a memory-access bandwidth close to the theoretical bandwidth. However, even if the memory-access bandwidth is low and some loops report low occupancy, the kernel could be optimal [3].

3.3.3.1 High Stall Percentage

High stall percentage implies that the pipeline is stalled due to memory-access bottleneck or unavailability of channel data. We explain why the stalls are caused and how to overcome those.

- Stalls are caused by unavailability of global memory data due to the following reasons.
  - Accessing a lot of data in one clock-cycle causes a pipeline to be stalled. If the occupancy is also high, you have to reduce the global memory access. This can be done by temporally storing frequently-accessed data in the local memory. You can also use memory interleaving to distribute the memory access equally to all memory banks.
  - Using multiple memory accesses for small data may also stall the pipeline. You have to check whether the memory access is coalesced since non-coalesced access could cause inefficient bandwidth utilization.

  Improving the global-memory access is discussed in Sect. 5.3.
- Stalls are also caused by the unavailability of channel data. This can happen due to the imbalances between read and write operations. For example, if the stall percentage of a write-channel operation is high, the occupancy and activity of the read-channel operation could be low. In such cases, the speed of the read-

channel operation is too slow compared to the write channel call. Improving the channel access is discussed in Sect. 5.7.
- You can ignore the case of "high-stall and low-occupancy" percentages. In this case, the memory-access operation is belonging to a non-critical path.

3.3.3.2 Occupancy

Low Occupancy in Non-critical Paths

Low occupancy could occur when a memory or a channel operation is in a non-critical path. For example, Listing 3.1 shows memory accesses in critical and non-critical paths. In this code, the two inner-loops are executed in parallel in a pipelined manner. The first loop is in the critical path since it has 100 loop-iterations. The second loop with only ten loop-iterations is in a non-critical path. Therefore, the memory access to arrays `a` and `c` should give a large occupancy whereas the rest should give a small occupancy.

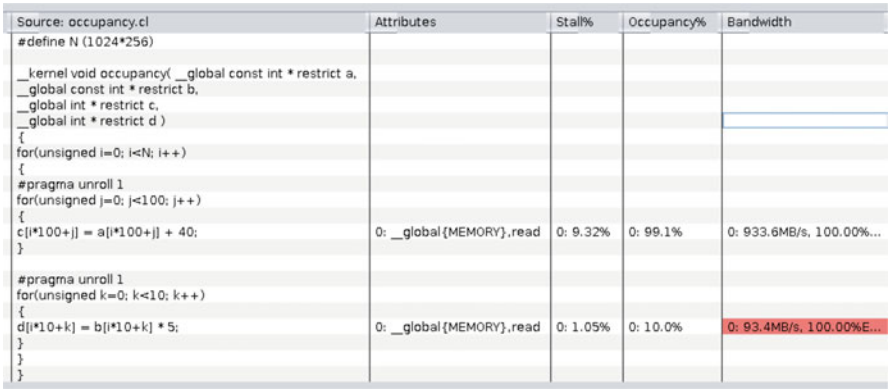
```
#define N (1024*256)

__kernel void occupancy(  __global const int * restrict a,
              __global const int * restrict b,
              __global int * restrict c,
              __global int * restrict d )
{
  for(unsigned i=0; i<N; i++)
  {
    #pragma unroll 1
    for(unsigned j=0; j<100; j++)
    {
      c[i*100+j] = a[i*100+j] + 40;
    }

    #pragma unroll 1
    for(unsigned k=0; k<10; k++)
    {
      d[i*10+k] = b[i*10+k] * 5;
    }
  }
}
```

Listing 3.1 Occupancy in critical and non-critical paths

Figure 3.9 shows the profiling result. The occupancy values shown inside the critical and non-critical paths are 99.1% and 10%, respectively. Since the data are not accessed frequently in the non-critical paths, the occupancy is small. This low occupancy value is not a matter of concern.

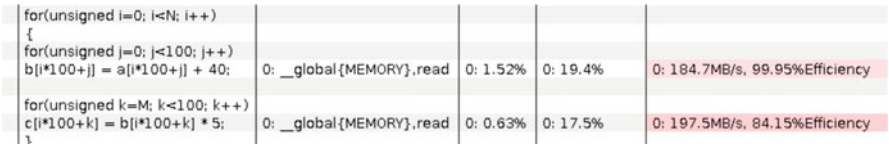
No Stall, Low Occupancy, and Low Bandwidth

If a memory operation is in a critical path of the kernel code and the occupancy is low for that memory operation, the reason could be the serial execution caused by data dependencies. The code in Listing 3.2 has a memory dependency in array



Source: occupancy.cl	Attributes	Stall%	Occupancy%	Bandwidth
#define N (1024*256)				
__kernel void occupancy(__global const int * restrict a,				
__global const int * restrict b,				
__global int * restrict c,				
__global int * restrict d)				
{				
for(unsigned i=0; i<N; i++)				
{				
#pragma unroll 1				
for(unsigned j=0; j<100; j++)				
{				
c[i*100+j] = a[i*100+j] + 40;	0: __global{MEMORY},read	0: 9.32%	0: 99.1%	0: 933.6MB/s, 100.00%...
}				
#pragma unroll 1				
for(unsigned k=0; k<10; k++)				
{				
d[i*10+k] = b[i*10+k] * 5;	0: __global{MEMORY},read	0: 1.05%	0: 10.0%	0: 93.4MB/s, 100.00%E...
}				
}				
}				

Fig. 3.9 The profiling result that shows the occupancy in critical and non-critical paths



for(unsigned i=0; i<N; i++)				
{				
for(unsigned j=0; j<100; j++)				
b[i*100+j] = a[i*100+j] + 40;	0: __global{MEMORY},read	0: 1.52%	0: 19.4%	0: 184.7MB/s, 99.95%Efficiency
for(unsigned k=M; k<100; k++)				
c[i*100+k] = b[i*100+k] * 5;	0: __global{MEMORY},read	0: 0.63%	0: 17.5%	0: 197.5MB/s, 84.15%Efficiency
}				

Fig. 3.10 Profiling result of the code in Listing 3.2

b. This will result in a serial execution, so that the occupancy and the bandwidth decreases as shown in Fig. 3.10. How to solve this problem is explained in Sect. 4.4.

```
#define N (1024*1)
__kernel void memdependency(  __global const int * restrict a,
                __global int * restrict b,
                __global int * restrict c,
                int M )
{
  for(unsigned i=0; i<N; i++)
  {
    for(unsigned j=0; j<100; j++)
      b[i*100+j] = a[i*100+j] + 40;

    for(unsigned k=M; k<100; k++)
      c[i*100+k] = b[i*100+k] * 5;
  }
}
```

Listing 3.2 Serial execution due to global memory dependency

No Stall, High Occupancy, and Low Bandwidth

Figure 3.11 shows a compilation report that has no stall, high occupancy, and low bandwidth with high efficiency. Since there is no stall and the occupancy is high, you can say that the kernel performs well. However, if the memory-access bandwidth is low, there is a lot of room for more data access. Therefore, you may increase the

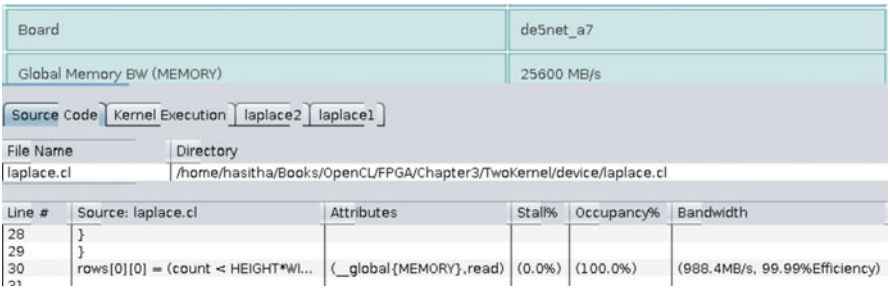


Fig. 3.11 No stall, high occupancy, low bandwidth with high efficiency

degree of parallelism by accessing and processing more data in parallel. This can be done by using SIMD vectorization or loop-unrolling. You can also use more compute units or more concurrent kernels. However, you need more resources to do more parallel operations. How to increase the degree of parallelism is explained in Chap. 4.

3.3.3.3 Low Bandwidth Efficiency

Low bandwidth efficiency occurs when the amount of useful data are smaller compared to the data accessed. This usually happens in cases such as random access, stride access, etc. To increase the efficiency, you have to access consecutive memory addresses.

3.4 Execution Phase

In order to obtain the configuration data for execution, you have to do the second stage compilation. You can do the second stage compilation using the following command.

```
aoc <intermediate_binary>.aoco -o <bit-stream>.aocx
```

The input is the <intermediate_binary>.aoco file generated in the first stage. The output is the <bit-stream>.aocx file which is used to configure the FPGA. You can do a full compilation (the first stage and second stage combined) as follows using the <kernel_code>.cl file as the input.

```
aoc <kernel_code>.cl -o <bit-stream>.aocx
```

After the kernel is compiled, you can execute it on an FPGA-based computing system and evaluate the processing time. You can also evaluate the actual resource usage and clock frequency by using the acl_quartus.log file. This file is generated after the second stage compilation. If you are not satisfied with the performance or

Table 3.1 Compilation reports in different SDK versions

Compilation stage	Available information	Description
Emulation		Behavior of the code
First stage	Loop analysis	Pipelined information, initiation interval, unrolling information, etc.
	Area analysis of system	Estimated resource utilization of each block
	Area analysis of source	Estimated resource utilization of each kernel code line
	System viewer	Structure of the FPGA accelerator
Second stage	Area analysis	Actual resource utilization
	Clock frequency	The clock frequency of the accelerator
Profiling	Profiling results	Memory access information

resource utilization, you can re-write the kernel code as shown in Fig. 3.1. Note that, do not use kernels compiled with `--profile` option for the final execution since they may have a significantly low clock frequency.

3.5 Summary

Table 3.1 shows the summary of the reports available at different stages.

References

1. Intel FPGA SDK for OpenCL (2017). https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html
2. Intel FPGA SDK for OpenCL, Programming guide (2017). https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf
3. Intel FPGA SDK for OpenCL, Best practices guide (2017). https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf
4. Z. Wang, B. He, W. Zhang, S. Jiang, A performance analysis framework for optimizing OpenCL applications on FPGAs, in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Barcelona (2016), pp. 114–125
5. H.R. Zohouri, N. Maruyamay, A. Smith, M. Matsuda, S. Matsuoka, Evaluating and optimizing OpenCL Kernels for high performance computing with FPGAs, in *International Conference for High Performance Computing, Networking, Storage and Analysis* (2016), pp. 409–420

Chapter 4
FPGA-Oriented Parallel Programming

4.1 Overview of NDRange Kernels and Single-Work-Item Kernels

OpenCL for FPGA uses two types of kernels: NDRange kernels and single-work-item kernels [1, 2]. Single-work-item kernels are also called task kernels. An NDRange kernel has many work-items whereas a single-work-item kernel has only one. There are two major differences between NDRange and single-work-item kernels. The first difference is in programming styles. The NDRange kernels use a GPU-like programming style, where multiple work-items are processed in parallel using global and local work-item IDs. The single-work-item kernels use a CPU-like programming style, where only one work-item is processed throughout the kernel execution. The second difference is in data sharing methods. The NDRange kernels share data among multiple work-items by using a local memory. The single-work-item kernels share data among multiple loop-iterations by using a private memory. Recently, there are many works for FPGA accelerators based on the NDRange kernels [3–5] and also the single-work-item kernels [6, 7].

4.1.1 NDRange Kernel

An NDRange kernel is executed by multiple work-items. The work-items are launched one after another by one cycle apart and processed in a pipeline manner. Listing 4.1 shows an NDRange kernel code for vector addition. The offline compiler generates a pipelined datapath for this kernel as shown in Fig. 4.1a. Figure 4.1b shows the timechart of the computation. In the first cycle, work-item 1 is launched and it loads data from the memory. In the next cycle, work-item 2 is launched and it loads data while work-item 1 performs the addition. In the next cycle, work-item 3 is launched and it loads data. At the same time, work-item 2 performs the addition

H.M. Waidyasooriya et al., *Design of FPGA-Based Computing Systems with OpenCL*, https://doi.org/10.1007/978-3-319-68161-0_4

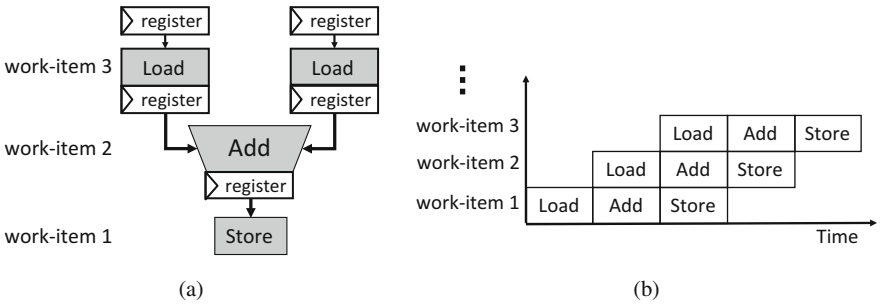


Fig. 4.1 Vector addition for an NDRange kernel. (**a**) Pipelined datapath. (**b**) Timechart

while work-item 1 stores data. Therefore, three work-items are executed in parallel in different stages.

```
__kernel void vectoradd_NDRange ( __global const float *a,
                                  __global const float *b,
                                  __global float *answer)
{
    int tid = get_global_id(0);
    answer[tid] = a[tid] + b[tid];
}
```

Listing 4.1 OpenCL kernel code for vector addition

As you can see in Fig. 4.1b, the way of execution is quite different from that of GPU's [8], although the NDRange-kernel code is very similar to a GPU code. In GPUs, work-items are launched simultaneously and executed in a single-instruction multiple-data (SIMD) manner. That is, multiple work-items execute the same operation simultaneously for different data. On the other hand, in FPGAs, work-items are launched and executed one after another in a pipelined manner. As a result, work-items perform different operations simultaneously for different data. Therefore, the behavior of the NDRange kernel is similar to multiple-instructions multiple-data (MIMD) computation, and very different from that of GPU's. Note that it is possible to use SIMD computation in FPGAs and this is discussed in Sect. 4.3.2.

When there are data dependencies between work-items, their data should be shared. The data sharing can be done through local or global memories. You may have to use barrier functions to ensure the correct execution order. All work-items in a work-group must execute this function before they are allowed to continue execution beyond the barrier. Since FPGAs use pipelines, the penalty for waiting until all work-items execute the barrier function is very large compared to GPUs.

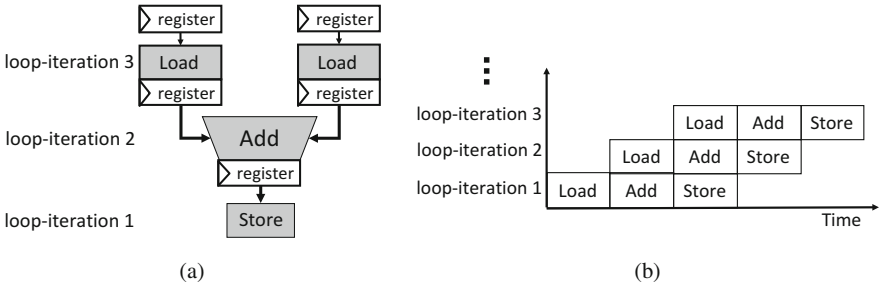


Fig. 4.2 Vector addition for a single-work-item kernel. (**a**) Pipelined datapath. (**b**) Timechart

4.1.2 Single-Work-Item Kernel

The implementation of a single-work-item kernel is very similar to that of a C-program. A single-work-item kernel contains loops, each of which has many loop-iterations. Multiple loop-iterations are computed in different pipeline stages in parallel, which is called loop-pipelining. As a result, we can utilize pipeline-level parallelism.

Listing 4.2 shows a code of single-work-item kernel for vector addition. For the single-work-item kernel, loop-iterations are used as the unit of execution of a kernel. The offline compiler generates a pipelined datapath as shown in Fig. 4.2a. Figure 4.2b shows the timechart of the computation. In the first cycle, loop-iteration 1 is launched and it loads data from the memory. In the next cycle, loop-iteration 2 is launched and it loads data while loop-iteration 1 performs the addition. In the next cycle, loop-iteration 3 is launched and it loads data. At the same time, loop-iteration 2 performs the addition while loop-iteration 1 stores data. At this stage, three loop-iterations are executed in parallel at different pipeline stages.

```
1 __kernel void vectoradd_single_work_item ( __global const float *a,
2                                            __global const float *b,
3                                            __global float *answer)
4 {
5     for(int i=0, i<SIZE; i++)
6       answer[i] = a[i] + b[i];
7 }
```

Listing 4.2 Single-work-item kernel code for vector addition

Although the codes of the single-work-item and NDRange kernel codes are different, their datapaths and the ways of execution are very similar as shown in Figs. 4.1 and 4.2. This is because there is no data dependency between work-items of the NDRange and between loop-iterations of the single-work-item kernel. If there are data dependencies, the behaviors of the two types of kernels are different. For the single-work-item kernels, the programmer need not do anything special to preserve the data dependency. For the NDRange kernels, barrier functions are required to preserve data dependencies among work-items as explained in Sect. 4.1.1.

4.1.3 Summary of Differences

In an NDRange kernel, work-items are executed in parallel in a pipeline manner. In a single-work-item kernel, loop-iterations are executed in parallel in a pipeline manner. When there is no data dependency, both kernels provide very similar results. When there are data dependencies, single-work-item kernels are easy to use and probably provide a better performance compared to NDRange kernels.

You can use NDRange kernels in the following situations.

- When there are no data sharing between work-items.
- When there is a very specific data sharing pattern such as the one in DFT (discrete Fourier Transformation) [9].
- When you want to use the same kernel code in both FPGAs and GPUs.

You can use a single-work-item kernels in the following situations.

- When there are data dependencies.
- When you want to port a typical CPU code to an FPGA.

Note that, single-work-item kernels work extremely poorly on GPUs. Since there is only one work-item, the code is executed in serial using just one GPU core.

4.2 Performance-Improvement Techniques Common to Single-Work-Item and NDRange Kernels

4.2.1 When and Where to Use Loop Unrolling

Loop unrolling is a very important method to increase the degree of parallelism. Unrolling loops allows more data to be processed in one clock cycle. It reduces the number of loop-iterations at the expense of increased resource utilization. To unroll a loop, `#pragma unroll <N>` directive is used, where the unroll factor is given by `<N>`. If the unroll factor is not specified, the offline compiler tries to unroll the loop completely. It is important to keep in mind that the offline compiler may unroll loops, even when there is no `#pragma unroll` directive.

Implementation of loop unrolling on FPGAs is very different from that of GPUs. In FPGAs, unrolling is not just removing the loop index operations. It significantly changes the structure of a kernel by applying more parallel operations. As a result, although the performances are increased, the resource utilization can be significantly large.

Listing 4.3 shows an example of loop-unrolling. The inner-loop is completely unrolled. As a result, the eight loop-iterations in the inner-loop can be processed in parallel. Table 4.1 shows the comparison of the processing times using loop-unrolling and without loop-unrolling. The processing time is decreased to 11.25% when using loop-unrolling with the almost same clock frequency.

```
__kernel void myKernel( __global const float * restrict din,
                        __global float * restrict dout )
{
  for(int i=0; i<iteration; i++)
  {
    float accum = 0;
    #pragma unroll
    for(int j=0; j<8; j++)
    {
      accum += din[i*8+j];
    }
    dout[i] = accum * K;
  }
}
```

Listing 4.3 Loop-unrolling

Table 4.1 Comparison of the processing times using loop-unrolling and without using loop-unrolling

Implementation	Processing time (ms)	Clock frequency (MHz)
Without using loop-unrolling	32.0	297
Using loop-unrolling	3.6	303

When unrolling loops, the following points should be considered.

- Loop-unrolling does not only increase the resource utilization, but also increases the required memory bandwidth. Therefore, you have to reduce the unroll factor if the kernel is memory-bound.
- Using large unroll factors may reduce the clock frequency due to large resource utilization. You have to consider reducing the unroll factor if the clock frequency is low.
- You have to stop automatic unrolling if the resource utilization is too high. This can be done by using `#pragma unroll <N>` directive. If the unroll factor is 1, the loop will not be unrolled.
- Unrolling the outer-loops of a nested-loop structure may increase the resource utilization significantly. Therefore, it is better to unroll the inner-loops first.

The offline compiler may not implement loop-unrolling in the following situations.

- When the kernel does not fit into the FPGA due to large resource utilization.
- When the loop boundaries are not constants.
- When loops contain complex control flows that are unknown at the compilation.

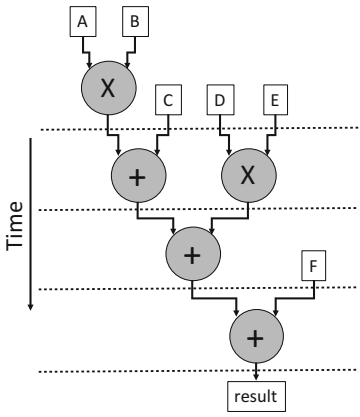


Fig. 4.3 Scheduled DFG for Listing 4.4

4.2.2 Optimizing Floating-Point Operations

4.2.2.1 Re-ordering Floating-Point Operations

The offline compiler performs the floating-point operations exactly as those are mentioned in the code. As a result, the implementation is consistent with IEEE standard 754-2008. The final results match the CPU implementation of the same code. The scheduled DFG in Fig. 4.3 shows the floating-point computation for Listing 4.4. The latency of the computation is four clock cycles.

```
1 __kernel void myKernel( __global float * A, ... )
2 {
3     result = (((A * B) + C) + (D * E)) + F;
4 }
```

Listing 4.4 Example of a floating-point computation

More efficient hardware can be implemented by a balanced-tree of floating-point operations as shown in Fig. 4.4. This is done by using the compile option `--fp-relaxed` as shown below.

```
aoc --fp-relaxed <kernel_file>.cl
```

In terms of latency, this implementation is more efficient than the one shown in Fig. 4.3. However, the computed results could be different in two implementations due to the re-order of floating-point operations.

4.2.2.2 Using Fused Floating-Point Operations

The offline compiler applies rounding operations according to the IEEE standard 754-2008. However, this may require a significant amount of hardware resources. If an application can tolerate small differences in floating-point results, you can reduce the rounding operations by using compile option `--fpc` for fused floating-point operations as follows.

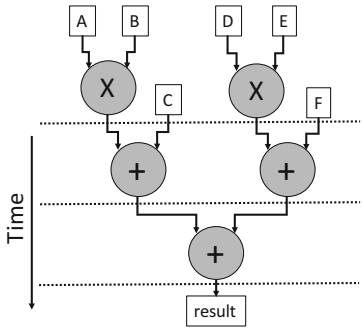


Fig. 4.4 Balanced tree of floating-point operations implemented using `--fp-relaxed` compiler option

```
aoc --fpc <kernel_fileneme>.cl
```

This option directs the offline compiler to round a floating-point operation only at the end of the computation. At the intermediate steps, results with additional precision bits are used for computation. At the final step, the additional precision bits are removed to round the results. This method may not consistent with IEEE standard 754-2008, but could have a better numerical accuracy.

4.2.3 Optimizing Fixed-Point Operations

Fixed-point computation uses less resources compared to floating-point computation. Therefore, it is better to use fixed-point computations whenever possible. The available data types for fixed-point computation are only `char` (8-bit), `short` (16-bit), `int` (32-bit), and `long` (64-bit). Therefore, it is not possible to represent data types with arbitrary widths; for example, 17-bit data. However, you can use bit-masks to direct compiler to perform optimization so as to implement desired type of hardware. Listing 4.5 allows the offline compiler to implement a 17-bit adder to add two 17-bit data. The result is also kept at 18-bit wide.

```
__kernel void myKernel( ... )
{
    unsigned int res;
    res = ((0x1FFFF & A[i]) + (0x1FFFF & B[i])) & 0x3FFFF;
    dout[i] = res & 0x3FFFF;
}
```

Listing 4.5 Optimizing fixed-point operations using bit masks

4.2.4 Optimizing Vector Operations

Vector operations, using built-in vector types are very important to coalesce the memory access as described in Sect. 5.3. When you update one element of a vector

type, update all the elements. If such update is not required, you can update using dummy values as shown in Listing 4.6. This results in a simple hardware structure, and reduces the resource utilization.

```
__kernel void update (__global const float4 * restrict din,
                      __global const float4 * restrict dout)
{
  int tid = get_global_id(0);
  dout[tid].x = din[tid].x + ...;
  dout[tid].y = din[tid].y - ...;
  dout[tid].z = din[tid].z + ...;
  dout[tid].w = 0.0f; //Update w even if that variable is not required.
}
```

Listing 4.6 Optimizing vector operations

4.2.5 Other Optimization Techniques

Insert the `restrict` keyword in pointer arguments whenever possible. This prevents offline compiler from creating unnecessary memory dependencies between non-conflicting load and store operations. The restrict keyword informs the offline compiler that the pointer is not alias other pointers. For example, if your kernel has two pointers to global memory, `A` and `B` that never overlap each other, declare the kernel as follows.

```
__kernel void myKernel (__global int * restrict A,
                        __global int * restrict B)
```

Note that inserting the restrict keyword on a pointer that aliases other pointers might result in incorrect results.

Avoiding expensive operations such as integer divisions and modulo operations could reduce the resource utilization. Modulo operations are often used to compute x, y, z coordinates. Instead of doing modulo computation, you can use multiple counters to compute x, y, z coordinates. Most floating-point operations except addition, multiplication, absolute value, and comparison are expensive. If the expensive functions are called only a few times in a kernel, you can pre-calculate those data and store in the memory.

4.3 Performance-Improvement Techniques for NDRange Kernels

4.3.1 Specifying the Work-Group Size

You can specify the work-group size in the kernel code to let the offline compiler perform aggressive optimization. There are two ways to specify the work-group size. One way is to specify a "maximum work-group size." The other way is to specify a "required work-group size."

4.3.1.1 How to Specify the Maximum Work Group Size

The maximum work-group size is specified by using the attribute `max_work_group_size(N)`, where `N` is the number of work-items. Example in Listing 4.7 sets the maximum work-group size to 128.

```
__attribute__((max_work_group_size(128)))
__kernel void vectoradd( __global const float * restrict a,
                         __global const float * restrict b,
                         __global float * restrict c )
{
  int tid = get_global_id(0);
  c[tid] = a[tid] + b[tid];
}
```

Listing 4.7 Specifying the maximum work-group size

In order to execute this kernel, you should do the following in the host code. Argument `local_work_size` of function `clEnqueueNDRangeKernel` must be set to a value smaller than or equal to the maximum work-group size. Otherwise, the call of function `clEnqueueNDRangeKernel` fails with reporting the error `CL_INVALID_WORK_GROUP_SIZE`.

4.3.1.2 How to Specify the Required Work-Group Size

The required work-group size is specified by using the attribute `reqd_work_group_size(X, Y, Z)`. The example in Listing 4.8 sets the required work-group size to (64,1,1).

```
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void vectoradd( __global const float * restrict a,
                         __global const float * restrict b,
                         __global float * restrict c )
{
  int tid = get_global_id(0);
  c[tid] = a[tid] + b[tid];
}
```

Listing 4.8 Specifying the required work-group size

In order to execute the kernel, you should do the following in the host code. Argument `local_work_size` of function `clEnqueueNDRangeKernel` in the host code must be set to the same value specified by the required work-group size. Otherwise, the call of function `clEnqueueNDRangeKernel` fails with reporting the error `CL_INVALID_WORK_GROUP_SIZE`.

4.3.2 Kernel Vectorization

To achieve a higher throughput, you can use kernel vectorization. It allows multiple work-items to be executed in the SIMD manner. In kernel vectorization, the offline

compiler combines multiple scalar operations (addition, multiplication, etc.) into one vector operation. Since more operations are done simultaneously in the SIMD manner, more resources are required. In addition, it may require more data per clock cycle and increases the required memory bandwidth. When you keep increasing the SIMD size, the kernel becomes memory-bound and the performance is saturated.

4.3.2.1 Automatic Vectorization

Using `num_simd_work_items(N)` attribute, as shown in Listing 4.9, allows the offline compiler to vectorize `N` operations of the same type. It must be used together with `reqd_work_group_size` attribute. The SIMD size must be a factor of 2 (that is, 2, 4, 8, 16, etc.), and the number of required work-items must be divisible by the SIMD size. Listing 4.9 shows an example of automatic vectorization with the four SIMD lanes. Work-items are evenly distributed among SIMD lanes. This kernel performs four times more work compared to a kernel without vectorization.

```
__attribute__((num_simd_work_items(4)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void myKernel( __global const float * restrict a,
                        __global const float * restrict b,
                        __global float * restrict c )
{
  int tid = get_global_id(0);
  c[tid] = a[tid] + b[tid];
}
```

Listing 4.9 Automatic vectorization

Since four load operations access consecutive locations in the global memory, the offline compiler coalesces the four loads into a single wider vector load. This optimization reduces the number of accesses to memory and potentially leads to better memory access patterns. If the memory access is data dependent or unknown at the compilation, the offline compiler will not coalesce the memory accesses.

4.3.2.2 Manual Vectorization

Listing 4.10 shows how to vectorize a kernel manually. The kernel performs four additions in four SIMD lanes. Therefore, the required number of work-items is only 1/4 compared to a kernel without vectorization. As a result, you have to modify the host code to use such number of work-items.

```
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void myKernel( __global const float * restrict a,
                        __global const float * restrict b,
                        __global float * restrict c )
{
  int tid = get_global_id(0);
  c[tid * 4 + 0] = a[tid * 4 + 0] + b[tid * 4 + 0];
  c[tid * 4 + 1] = a[tid * 4 + 1] + b[tid * 4 + 1];
  c[tid * 4 + 2] = a[tid * 4 + 2] + b[tid * 4 + 2];
  c[tid * 4 + 3] = a[tid * 4 + 3] + b[tid * 4 + 3];
}
```

Listing 4.10 Manual vectorization of a kernel

4.3.2.3 Using Vector Data Types

You can use vector data types as shown in Listing 4.11 to increase the computations per clock cycles. The number of memory accesses are reduced by coalescing four load (and store) operations into one load (and store) operation. Memory access coalescing is explained in detail in Sect. 5.3.1.

```
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void vectoradd( __global const float4 * restrict a,
                         __global const float4 * restrict b,
                         __global float4 * restrict c )
{
  int tid = get_global_id(0);
  c[tid] = a[tid] + b[tid];

   //instead you can use the following notation also
   /*
   c[tid].s0 = a[tid].s0 + b[tid].s0;
   c[tid].s1 = a[tid].s1 + b[tid].s1;
   c[tid].s2 = a[tid].s2 + b[tid].s2;
   c[tid].s3 = a[tid].s3 + b[tid].s3;
   */
}
```

Listing 4.11 Using vector data types

4.3.3 Increasing the Number of Compute Units

Multiple compute units can be used to achieve a higher throughput. The offline compiler implements each compute unit as a unique pipeline. Multiple compute units execute multiple work-groups simultaneously. Since a work-group is executed in one compute unit, you have to declare enough work-groups in the host code to utilize all compute units. The hardware scheduler in the FPGA dispatches work-groups to available compute units automatically.

The number of compute units is set by `num_compute_units(N)` attribute as shown in Listing 4.12. Increasing the number of compute units increases the resource usage and required memory bandwidth.

```
__attribute__((num_compute_units(4)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void MyKernel( __global const float * restrict a,
                        __global const float * restrict b,
                        __global float * restrict c )
{
  int tid = get_global_id(0);
  c[tid] = a[tid] + b[tid];
}
```

Listing 4.12 Implementing multiple compute units

Table 4.2 shows the comparison of processing times of different optimization methods. The attribute `reqd_work_group_size` is set to (64, 1, 1) and the same NDRange is used for all methods. We achieved nearly four times speed-up for

Table 4.2 Processing time comparison of optimization methods for NDRange kernels

Implementation	Optimization	Processing time (ms)	Clock frequency (MHz)
Listing 4.8	No optimization	884.7	303
Listing 4.9	Automatic vectorization by 4	227.1	304
Listing 4.10	Manual vectorization by 4	224.5	304
Listing 4.11	Using the vector data type `float4`	224.4	304
Listing 4.12	Using four compute units	546.8	302

different SIMD vectorization methods compared to no optimization. However, the speed-up using four compute units is just 1.6 times. The reason is the non-coalesced memory access when using multiple compute units. Since multiple compute units operate independently, all units compete for global memory access. This may lead to non-coalesced memory access.

4.3.4 Combining Kernel Vectorization with the Use of Multiple Compute Units

Both kernel vectorization and the use of multiple compute units increase the throughput by doing more computations and accessing more data in one clock cycle. The kernel vectorization increases the number of SIMD operations within a single compute unit, as shown in Fig. 4.5a. The work-items of the same work-group are executed in the SIMD manner. The SIMD operations use the same instructions so that the control logic is shared. Since only one compute unit is used, there is a one wide data-path to the global memory. Figure 4.5b shows the use of multiple compute units. Two compute units execute two work-groups simultaneously. Each compute unit has its own data-path to access memory.

The following are the advantages of using kernel vectorization compared to using multiple compute units.

- Resource utilization is smaller since the control logic is shared.
- Memory access is efficient and coalesced. This reduces the number of memory accesses.

Following is a disadvantage of using kernel vectorization compared to multiple compute units.

- SIMD vectorization size is limited to a multiple of 2. As a result, the resource usage may not be fully utilized.

In order to maximize the performance, you can combine kernel vectorization with the use of multiple compute units to fully utilize the resources. Let us consider an example of a kernel that has eight SIMD lanes, which performs eight operations per

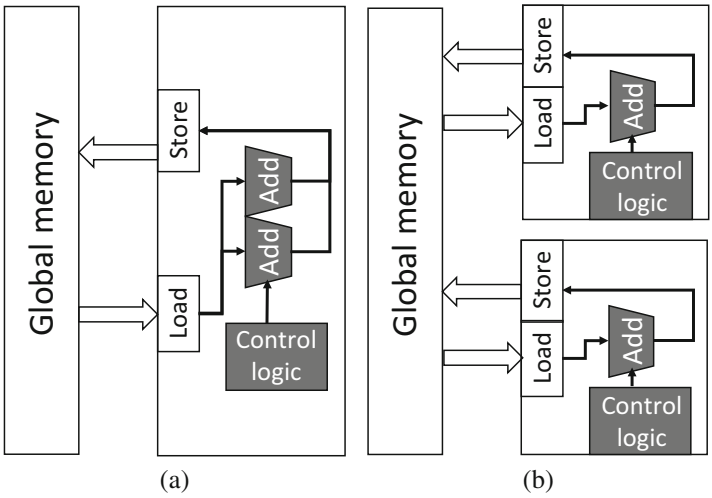


Fig. 4.5 Kernel vectorization vs. using multiple compute units. (**a**) Kernel vectorization by two. (**b**) Using two compute units

clock cycle. We assume that we cannot increase the number of SIMD lanes to 16 due to resource constraints. Instead, we can write a kernel code as shown in Listing 4.13 that uses three compute units where each has four SIMD lanes. The implementation of this kernel is shown in Fig. 4.6. This provides 12 operations per clock cycle. This implementation can perform more operations per clock cycle compared to the kernel with eight SIMD lanes.

```
__attribute__((num_simd_work_items(4)))
__attribute__((num_compute_units(3)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void vectoradd ( __global const float * restrict a,
                          __global const float * restrict b,
                          __global float * restrict c)
{
    size_t tid = get_global_id(0);
    c[tid] = a[tid] + b[tid];
}
```

Listing 4.13 Combining kernel vectorization with the use of multiple compute units

4.4 Performance-Improvement Techniques for Single-Work-Item Kernels

In order to improve the performance of a single-work-item kernel, the following should be done.

- Implement pipelines for all loops.

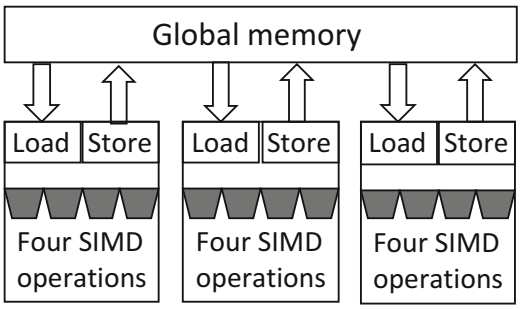


Fig. 4.6 Using both kernel vectorization and multiple compute units

```
Loop Report:

 + Loop "Block1" (file nestedloop.cl line 6)
 | Pipelined with successive iterations launched every 2 cycles due to:
 |
 |     Pipeline structure: every terminating loop with subloops has
iterations launched at least 2 cycles apart.
 |     Having successive iterations launched every two cycles should still
lead to good performance
 |     if the inner loop is pipelined well and has sufficiently high
number of iterations.
 |
 |
 |-+ Loop "Block2" (file nestedloop.cl line 8)
     Pipelined well. Successive iterations are launched every cycle.
```

Fig. 4.7 Compiler report for Listing 4.14

- Implement the kernel with the initiation interval (II) of 1, where the initiation interval is the loop-iteration-launch-delay.

As explained in Sect. 4.1, the single-work-item kernels implement loop-pipelining. In order to preserve the data dependencies among loop-iterations, the offline compiler could increase II or even reduce the clock frequency. This section explains how to avoid such situations, and how to optimize the kernel for maximum performance.

4.4.1 Avoiding Nested Loops

Listing 4.14 shows a nested-loop structure. Since the outer-loop indexes are used in the computation of the inner-loop, the offline compiler assumes that the outer-loop must wait until the inner-loop is finished. Figure 4.7 shows the compilation report for Listing 4.14. It shows that the outer-loop iterations are launched at least two cycles apart. Since there are `N` loop-iterations in the inner-loop, II of the outer-loop is $N + 2$ cycles. The extra two cycle delay is caused by the control overhead of the nested loops.

```
#define N (1024*16)

__kernel void nestedloop( __global const int * restrict din,
                          __global int * restrict dout )
{
  for(unsigned i=0; i<N; i++)
  {
    for(unsigned j=0; j<N; j++)
    {
      dout[i*N+j] = din[i*N+j] + 40;
    }
  }
}
```

Listing 4.14 A nested loop

The extra two cycle delay can be eliminated by re-arranging the loop structure into a single large loop as shown in Listing 4.15. According to the compilation report, loop-iterations are launched in every cycle.

```
#define N (1024*16)

__kernel void singleloop( __global const int * restrict din,
                          __global int * restrict dout )
{
  for(unsigned i=0; i<N*N; i++)
    dout[i] = din[i] + 40;
}
```

Listing 4.15 Re-arranging the nested-loop in Listing 4.14 into a single loop

Table 4.3 shows the comparison of the processing times of the nested-loop implementation (Listing 4.14) and single-loop implementation (Listing 4.15). The inner-loop of the nested-loop structure contains 16,384 loop-iterations. The comparison is done using Terasic DE5 FPGA board [10]. The kernels are compiled using Intel FPGA SDK for OpenCL 16.0. There is not much difference between the implementations. Since the clock frequencies of both implementations are the same, we can directly compare the number of clock cycles. Since the inner-loop contains a large number of loop-iterations, a delay of extra two clock cycles is negligible. If the inner-loop had only a few loop-iterations, we could have seen a large difference due to the two cycle delay.

It is possible to have a higher clock frequency for the single-loop implementation compared to that for the nested-loop implementation. Table 4.4 shows the comparison of the processing times of the nested-loop and single-loop implementations for the vector addition example shown in Listing 4.2. The single-loop implementation has a higher clock frequency and thus a smaller processing time compared to the nested-loop implementation. For larger kernels, the singe-loop implementation tends to have a higher frequency, so that it is better to avoid nested-loops.

Table 4.3 Comparison of the processing times of the nested-loop (Listing 4.14) and single-loop (Listing 4.15) implementations

Implementation	Processing time (ms)	Clock frequency (MHz)
Listing 4.14 (using nested loops)	887.8	302
Listing 4.15 (using a single loop)	887.7	302

Table 4.4 Comparison of the processing times of the nested-loop and single-loop implementations for the vector addition example shown in Listing 4.2

Implementation	Processing time (ms)	Clock frequency (MHz)
Vector addition (using nested loops)	1196.7	285
Vector addition (using a single loop)	881.6	304

```
Loop Report:

 + Loop "Block1" (file unrollinnerloop.cl line 7)
 | Pipelined well. Successive iterations are launched every cycle.
 |
 |
 |-+ Fully unrolled loop (file unrollinnerloop.cl line 10)
     Loop was fully unrolled due to "#pragma unroll" annotation.
```

Fig. 4.8 Compilation report for Listing 4.16

Loop-unrolling can be used to eliminate nested-loop structures. Listing 4.16 shows that the inner-loop is unrolled completely to eliminate the nested-loop structure. The compilation report in Fig. 4.8 shows that the loop-iterations are launched every cycle. If the number of iterations in the inner-loop is small, it is easy to unroll it. Keep in mind that loop-unrolling increases the resource utilization and the required memory bandwidth.

```
1  #define N (1024*32)
2  #define M 64
3
4  __kernel void unrollinnerloop( __global const int * restrict din,
5                                 __global int * restrict dout )
6  {
7    for(unsigned i=0; i<N; i++)
8    {
9      #pragma unroll 64
10     for(unsigned j=0; j<M; j++)
11     {
12       dout[i*M+j] = din[i*M+j] + 40;
13     }
14   }
15 }
```

Listing 4.16 Using loop-unrolling to eliminate nested-loops

For additional information, you can refer the work in [11] that explains how to predict the number of clock cycles in nested-loop structures. It uses the compilation report to find which loops are processed in parallel and which loops are processed in serial. Using this information and the number of loop-iterations in each loop, the number of clock cycles can be predicted.

4.4.2 Avoiding Serial Executions Due to Data Dependency

Listings 4.17 shows a kernel code with nested loops that computes the sum of integers. The computation in line 12 in the inner-loop uses the result of the computation in line 14. Similarly, the computation in line 14 uses the result of the computation in line 12. As a result, the computation in the inner-loop (from lines 10 to 13) and the computation in line 14 must be done in serial manner. The compilation report in Fig. 4.9 shows that the "loop-iterations are executed in serial in the data-dependent region."

```
1  #define N 1024
2
3  __kernel void unoptimized( __global const int * restrict A,
4                             __global const int * restrict B,
5                             __global int* restrict dout)
6  {
7    int sum = 0;
8    for (unsigned i = 0; i < N; i++)
9    {
10     for (unsigned j = 0; j < N; j++)
11     {
12       sum += A[i*N+j]
13     }
14     sum += B[i];
15   }
16   *dout = sum;
17 }
```

Listing 4.17 Serial execution due to data dependency

Listing 4.18 shows how to solve this data-dependency problem. The variable `sum` is removed from the inner-loop and a new variable `tmpsum` is used instead. Now, it is possible to overlap the computation in the inner-loop and the computation in line 15 in execution in a pipeline manner.

```
1  #define N 1024
2
3  __kernel void optimized( __global const int * restrict A,
4                           __global const int * restrict B,
5                           __global int* restrict dout)
6  {
7    int sum = 0;
8    for (unsigned i = 0; i < N; i++)
9    {
10     int tmpsum = 0;
11     for (unsigned j = 0; j < N; j++)
12     {
13       tmpsum += A[i*N+j]
14     }
15     sum += tmpsum;
16     sum += B[i];
17   }
18   *dout = sum;
19 }
```

Listing 4.18 Parallel execution after removing the data-dependency problem in Listings 4.17

The compilation report for Listing 4.18 is shown in Fig. 4.10. The serial execution region disappeared and the loop is successfully pipelined. In addition, iterations are launched at least two cycles apart according to the compilation report.

```
Loop Report:

 + Loop "Block1" (file datadependent.cl line 8)
 | Pipelined with successive iterations launched every 2 cycles due to:
 |
 |     Pipeline structure: every terminating loop with subloops has
iterations launched at least 2 cycles apart.
 |     Having successive iterations launched every two cycles should still
lead to good performance
 |     if the inner loop is pipelined well and has sufficiently high
number of iterations.
 |
 | Iterations executed serially across the region listed below.
 | Only a single loop iteration will execute inside the listed region.
 | This will cause performance degradation unless the region is pipelined
well
 | (can process an iteration every cycle).
 |
 |     Loop "Block2" (file datadependent.cl line 10)
 |     due to:
 |     Data dependency on variable sum  (file datadependent.cl line 7)
 |
 |
 |-+ Loop "Block2" (file datadependent.cl line 10)
     Pipelined well. Successive iterations are launched every cycle.
```

Fig. 4.9 Compilation report for the Listing 4.17

```
Loop Report:

 + Loop "Block1" (file nodatadependent.cl line 8)
 | Pipelined with successive iterations launched every 2 cycles due to:
 |
 |     Pipeline structure: every terminating loop with subloops has
iterations launched at least 2 cycles apart.
 |     Having successive iterations launched every two cycles should still
lead to good performance
 |     if the inner loop is pipelined well and has sufficiently high
number of iterations.
 |
 |
 |-+ Loop "Block2" (file nodatadependent.cl line 11)
     Pipelined well. Successive iterations are launched every cycle.
```

Fig. 4.10 Compilation report for Listing 4.18

This is because of the nested-loop structure explained in Sect. 4.4.1, and does not have any relationship with the data dependency.

Table 4.5 shows the comparison of the processing times for serial and parallel executions. Terasic DE5 FPGA board [10] and Intel FPGA SDK for OpenCL 16.0 are used for the evaluation. The processing time of Listing 4.18 is slightly smaller than that of Listing 4.17. Figure 4.11a shows the timechart for the computation in Listing 4.17. The computation in the inner-loop and the computations of line 14 are done in serial. The inner-loop has 1024 loop-iterations and has a long computation

Table 4.5 Comparison of the processing times for serial and parallel executions

Implementation	Processing time (ms)	Clock frequency (MHz)
Listing 4.17 (contain data dependency)	1130	239
Listing 4.18 (removed data dependency)	1083	247

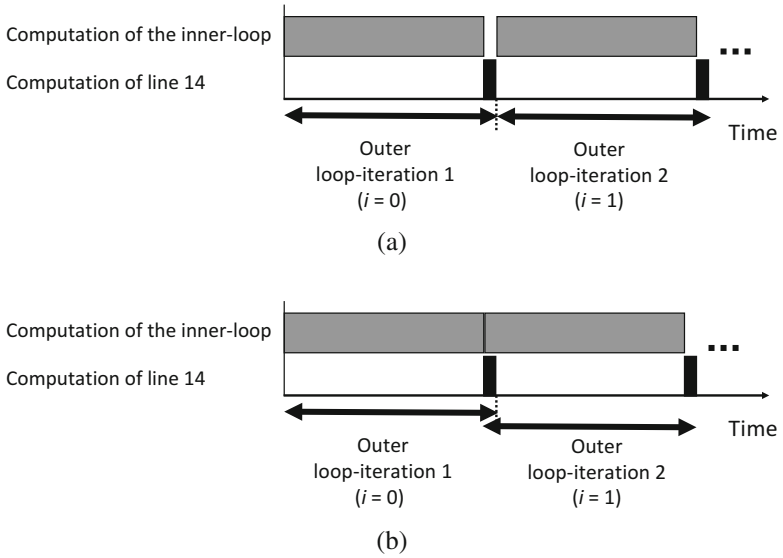


Fig. 4.11 Comparison of the timecharts. (**a**) Timechart for Listing 4.17. (**b**) Timechart for Listing 4.18

time compared to that of line 14, which is only one operation. Figure 4.11b shows the timechart for the computation in Listing 4.18. The computation in the inner-loop and the computations of line 15 are done in parallel.

Let us consider an another example. Listing 4.19 shows a nested-loop structure that has two inner-loops. Variable `sum` is updated in both loops. The computation in line 17 requires the result of the computation in line 12. Similarly, the computation in line 12 requires the results of the computation in line 17. Therefore, both inner-loops are executed in serial. Listing 4.20 shows how to solve this problem. Variable `sum` is removed in the inner-loops and new variables `tmpsum1` and `tmpsum2` are used. Therefore, the computations of the inner-loops do not require each other's results. As a result, the inner-loops are executed in parallel in pipeline manner.

```
1 #define N 1024
2
3 __kernel void unoptimized (__global const int * restrict A,
4                             __global const int * restrict B,
5                             __global int* restrict result)
6 {
7   int sum = 0;
8   for (unsigned i = 0; i < N; i++)
9   {
```

```
    for (unsigned j1 = 0; j1 < N; j1++)
    {
      sum += A[i*N+j1]
    }

    for (unsigned j2 = 0; j2 < N; j2++)
    {
      sum += B[i*N+j2]
    }
  }
  *result = sum;
}
```

Listing 4.19 Serial execution due to data dependency in two loops

```
#define N 1024

__kernel void optimized ( __global const int * restrict A,
                          __global const int * restrict B,
                          __global int * restrict dout)
{
  int sum = 0;
  for (unsigned i = 0; i < N; i++)
  {
    int tmpsum1 = 0;
    for (unsigned j1 = 0; j1 < N; j1++)
    {
      tmpsum1 += A[i*N+j1]
    }

    int tmpsum2 = 0;
    for (unsigned j2 = 0; j2 < N; j2++)
    {
      tmpsum2 += B[i*N+j2]
    }
    sum += tmpsum1;
    sum += tmpsum2;
  }
  *result = sum;
}
```

Listing 4.20 Parallel execution by removing data dependency in two loops

Table 4.6 shows the comparison of the processing times for serial execution (Listing 4.19) and parallel execution (Listing 4.20). Terasic DE5 FPGA board [10] and Intel FPGA SDK for OpenCL 16.0 are used for the evaluation. The processing time of Listing 4.20 is only 48% of that of Listing 4.19. Figure 4.12a shows the timechart for Listing 4.19. The computations in two inner-loops are done in serial. Figure 4.12b shows the timechart for Listing 4.20. The computations in two inner-loops are done in parallel. Since both inner-loops have the same number of loop-iterations, executing them in parallel leads to a large reduction of processing time.

4.4.3 Reducing Initiation Interval for Reduction Operations

The initiation interval is larger than one clock cycle for reduction operations due to data dependencies between loop-iterations. For an example of reduction operations, let us consider the summation of floating-point values shown in Listing 4.21. The

Table 4.6 Comparison of processing times for Listings 4.19 and 4.20

Implementation	Processing time (ms)	Clock frequency (MHz)
Listing 4.19 (contain data dependency among loops)	1912	283
Listing 4.20 (removed data dependency among loops)	935	287

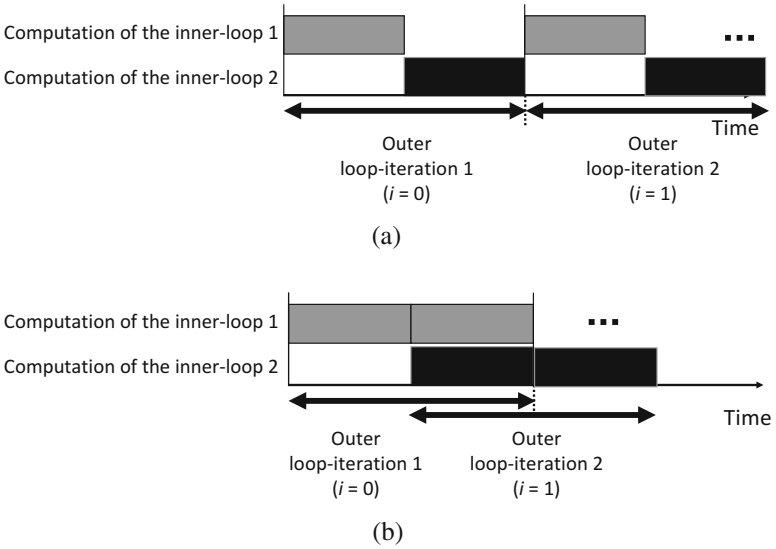


Fig. 4.12 Timecharts for Listings 4.19 and 4.20. (**a**) Timechart for Listing 4.19. (**b**) Timechart for Listing 4.20

variable `sum` is updated in each loop-iteration by adding `din[i]` to its current value. Therefore, each add operation requires the result of its previous operation. The latency of the floating-point addition is several clock-cycles. As a result, each add operation has to wait for several clock-cycles until the previous operation is finished.

```
#define N (1024*1024*16)
__kernel void unoptimized (__global const float * restrict din,
                           __global float * restrict dout )
{
  float sum = 0.0f;
  for (int i = 0; i < N; ++i)
  {
    sum += din[i];
  }
  *dout = sum;
}
```

Listing 4.21 Computation of the sum of floating-point values

```
Loop Report:

+ Loop "Block1" (file floatadd.cl line 7)
  Pipelined with successive iterations launched every 8 cycles due to:

      Data dependency on variable sum  (file floatadd.cl line 9)
      Largest Critical Path Contributor:
          96%: Fadd Operation  (file floatadd.cl line 9)
```

Fig. 4.13 Compilation report for Listing 4.21

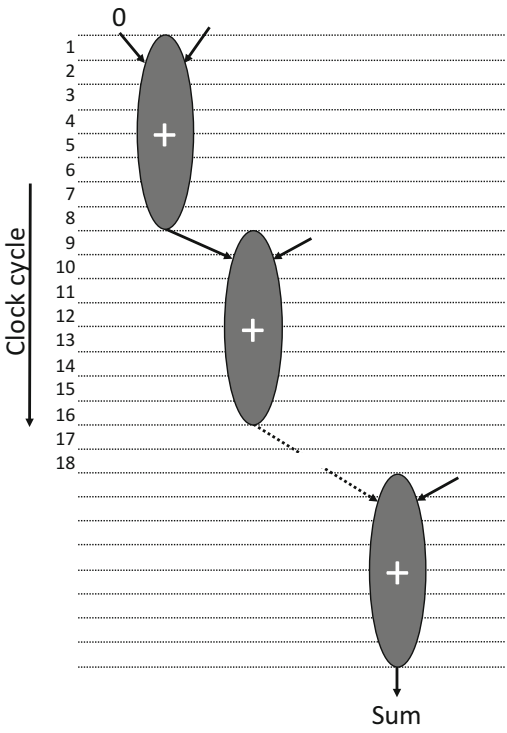


Fig. 4.14 Scheduled data-flow graph for Listing 4.21

The compilation report for Listing 4.21 is shown in Fig. 4.13. It reports that a loop-iteration is launched every eight clock cycles. In other words, the initiation interval is 8. Figure 4.14 shows the scheduled data-flow graph (SDFG) for Listing 4.21.

To reduce the initiation interval, you can use eight partial sums as shown in Fig. 4.15, and execute them in parallel as shown in Fig. 4.16. Listing 4.22 shows the kernel code corresponding to Fig. 4.16. Line 8 declares a shift-register to store the eight partial sums. In Lines 15–25, the partial sums are computed in parallel. In Lines 28–31, the results of partial sums are added together.

As the value of the number of partial sums in Fig. 4.15, we used the initiation interval of 8, which is obtained from the compilation report in Fig. 4.13. However, this way to determine the number of partial sums is not always the best. Table 4.7 shows the processing times for the different numbers of partial sums. Each row

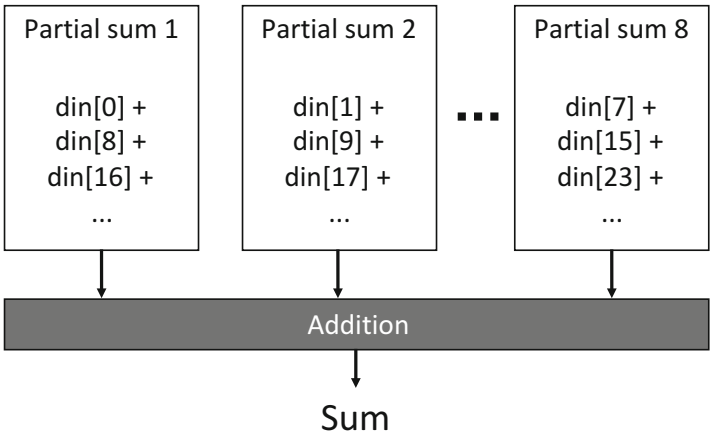


Fig. 4.15 Computation using eight partial sums

corresponds to the implementation with a certain number of partial sums. The processing time decreases as the number of partial sums becomes large. To obtain the best design, we should select the design with the minimum processing time.

```
#define N (1024*1024*16)
#define PARTIAL_SUMS 8

__kernel void optimized(  __global const float * restrict din,
              __global float * restrict dout )
{
  //Create shift-registers of length (PARTIAL_SUMS+1)
  float shift_reg[PARTIAL_SUMS+1];

  //Initiate all elements in shift registers to zero
  #pragma unroll 1
  for(int i=0; i<PARTIAL_SUMS+1; i++)
    shift_reg[i] = 0;

  for(int i=0; i<N; i++)
  {
    //compute PARTIAL_SUMS of partial sums
    shift_reg[PARTIAL_SUMS] = shift_reg[0] + din[i];

    #pragma unroll   //shift all elements simultaneously
    for(int j=0; j<PARTIAL_SUMS; j++)
    {
      shift_reg[j] = shift_reg[j+1];
    }
  }

  //add all partial sums together
  float sum = 0.0f;
  #pragma unroll //use 1 to prevent unroll to save resources
  for(unsigned i=0; i<PARTIAL_SUMS; i++)
    sum += shift_reg[i];

  *dout = sum;
}
```

Listing 4.22 Reduction operation using partial sums

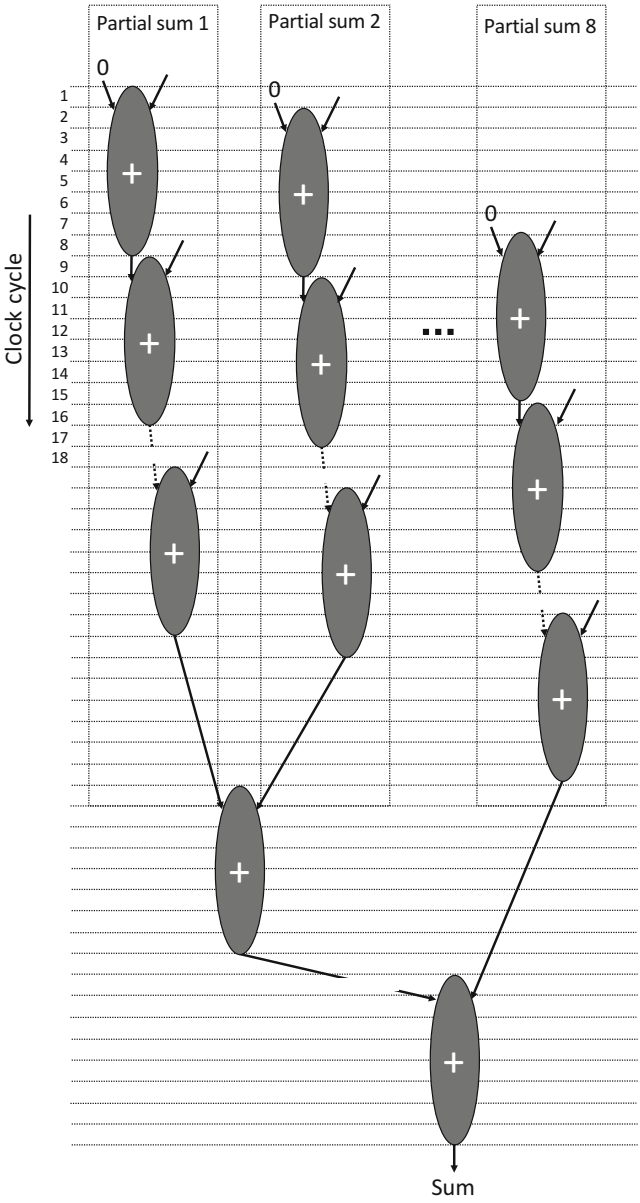


Fig. 4.16 Execution of the partial sums in parallel

Table 4.7 Comparison of processing times for different `PARTIAL_SUMS`

Implementation	Processing time (ms)	Clock frequency (MHz)	Initiation interval
No optimization	440.8	304	8 cycles
PARTIAL_SUMS=2	145.4	231	2 cycles
PARTIAL_SUMS=3	110.4	304	2 cycles
PARTIAL_SUMS=4	80.3	209	1 cycle
PARTIAL_SUMS=5	72.3	232	1 cycle
PARTIAL_SUMS=6	59.0	285	1 cycle
PARTIAL_SUMS=7	55.2	304	1 cycle
PARTIAL_SUMS=8	55.2	304	1 cycle
PARTIAL_SUMS=9	55.2	304	1 cycle
PARTIAL_SUMS=10	55.2	304	1 cycle

```
Loop Report:

+ Loop "Block1" (file unoptimized.cl line 6)
  Pipelined with successive iterations launched every 324 cycles due to:

      Memory dependency on Load Operation from: (file unoptimized.cl line 7)
        Store Operation (file unoptimized.cl line 7)
      Largest Critical Path Contributors:
          49%: Load Operation  (file unoptimized.cl line 7)
          49%: Store Operation  (file unoptimized.cl line 7)
```

Fig. 4.17 Compilation report for Listing 4.23

4.4.4 Reducing Initiation Interval Due to Read-Modify-Write Operations to Global Memory

Since the latency of the global memory access is very large, read-modify-write operations to the global memory cause significant performance degradation. Listing 4.23 shows an example of read-modify-write operations to the global memory. The compilation report for Listing 4.23 is shown in Fig. 4.17. The initiation interval is 324 cycles and very large.

```
1 #define N (1024*256)
2
3 __kernel void unoptimized(__global int * restrict dat,
4                __global const int * restrict randadrs)
5 {
6   for(int i=0; i<N; i++)
7     dat[i] = dat[randadrs[i]] + 11;
8 }
```

Listing 4.23 Kernel code for read-modify-write operations to the global memory

We can reduce the initiation interval by coping the global-memory data into the on-chip memory (local/private memory) since the on-chip memory has a significantly smaller latency compared to the global memory. After coping the

```
Loop Report:

 + Loop "Block1" (file optimized.cl line 8)
   Pipelined well. Successive iterations are launched every cycle.


 + Loop "Block2" (file optimized.cl line 11)
   Pipelined with successive iterations launched every 2 cycles due to:

       Memory dependency on Load Operation from: (file optimized.cl line 12)
         Store Operation (file optimized.cl line 12)
       Largest Critical Path Contributors:
           65%: Load Operation  (file optimized.cl line 12)
           34%: Store Operation  (file optimized.cl line 12)


 + Loop "Block3" (file optimized.cl line 14)
   Pipelined well. Successive iterations are launched every cycle.
```

Fig. 4.18 Compilation report of the code in Listing 4.24

Table 4.8 Comparison of processing times for Listings 4.23 and 4.24

Implementation	Processing time (ms)	Clock frequency (MHz)	Loop-iterations launch delay
Listing 4.23 (global memory dependency)	1952	296	324 cycles
Listing 4.24 (local memory dependency)	181	121	2 cycles

global-memory data, computation is done using the on-chip memory. The kernel code for this computation is shown in Listing 4.24. Figure 4.18 shows the compilation report, and the initiation interval is two.

```
1  #define N (1024*512)
2
3  __kernel void optimized(__global int * restrict dat,
4                          __global const int * restrict randadrs)
5  {
6    int tmp[N];
7
8    for(int i=0; i<N; i++)
9      tmp[i] = dat[i];
10
11   for(int i=0; i<N; i++)
12     tmp[i] = tmp[randadrs[i]] + 11;
13
14   for(int i=0; i<N; i++)
15     dat[i] = tmp[i];
16 }
```

Listing 4.24 Using the on-chip memory to reduce the initiation interval

Table 4.8 shows the processing time comparison. The improvements are significant since the initiation interval of 324 is reduced to only 2. Note that the offline compiler lowers the clock frequency significantly to launch loop-iterations every two cycles. In total, the processing time is reduced to 9%.

This method is applicable to the case where the data is small enough to be stored in the on-chip memory. For example, the size of the on-chip memory of the FPGAs

is a few tens of Mbytes even in the high-end FPGAs such as Stratix V and Arria 10. Therefore, only a small portion of data can be copied to the on-chip memory. However, you may repeat this process by re-using the same on-chip memory.

4.4.5 *Ignore Loop-Carried Dependencies Due to Read-Modify-Write Operations*

If the updated data are not required by another loop-iteration, next loop-iteration can be launched without waiting for the data of the previous iteration to be written to the global memory. In this case, you can instruct the offline compiler to ignore the dependencies by using `#pragma ivdep` directive as shown in Listing 4.25. The compilation report for Listing 4.25 is shown in Fig. 4.19. The initiation interval is one. Note that the programmer must be absolutely sure that there are no data dependencies when using **ivdep**. Otherwise, it will produce incorrect results.

```
#define N (1024*512)

__kernel void optimized(__global int * restrict dat,
                        __global const int * restrict randadrs)
{
  #pragma ivdep
  for(int i=0; i<N; i++)
    dat[i] = dat[randadrs[i]] + 11;
}
```

Listing 4.25 Data dependency due to global memory updates

Table 4.9 shows the comparison of the processing times for Listings 4.23, 4.24, and 4.25. The processing time for Listing 4.25 is significantly small compared to those for the other methods. The reasons for the processing-time reduction are the higher clock frequency and the small initiation interval of 1.

```
Loop Report:

 + Loop "Block1" (file optimized.cl line 7)
   Pipelined well. Successive iterations are launched every cycle.
```

Fig. 4.19 Compilation report of the code in Listing 4.25

Table 4.9 Processing time comparison for different

Implementation	Processing time (ms)	Clock frequency (MHz)	loop-iterations launch delay
Listing 4.23 (global memory dependency)	1952	296	324 cycles
Listing 4.24 (local memory dependency)	181	121	2 cycles
Listing 4.25 (using ivdep)	2	302	1 cycle

4.4.6 *Implementing a Single-Cycle Floating-Point Accumulator*

The implementation of the accumulation operation is quite different between older Stratix V FPGAs and the newer Arria 10 FPGAs. Listing 4.26 shows a simple accumulation operation. We compiled this code for Terasic DE5-Net FPGA Development Kit with Stratix V FPGA [10] and Terasic DE5a-Net Arria 10 FPGA Development Kit with Arria 10 FPGA [12]. The compilation reports are shown in Fig. 4.20. The compilation report for Stratix V FPGA is shown in Fig. 4.20a. The compilation is done using OpenCL 16.1 SDK and Quartus Prime standard 16.1 edition. As shown in the report, loop-iterations are launched every eight clock cycles. This is because of the data dependency in floating-point addition operation, that was explained in Sect. 4.4.3. The compilation report for Arria 10 FPGA is shown in Fig. 4.20b. The compilation is done using OpenCL 16.1 SDK and Quartus Prime Pro 16.1 edition. As shown in the report, despite the data dependencies, loop-iterations are launched every clock cycle. This is due to the **single-cycle accumulator feature** in Arria 10 FPGAs.

To use the single-cycle accumulator feature in Arria 10 FPGAs, the following conditions must be satisfied.

- Accumulation operation must be inside of a loop.
- Accumulator must be initialized to zero.

```
__kernel void accumulation( __global const float * restrict din,
                __global float * restrict dout,
                int N )
{
  float acc = 0.0f;
  for(int i=0; i<N; i++)
  {
    acc += din[i];
  }
  *dout = acc;
}
```

Listing 4.26 Simple floating-point accumulation

Listing 4.27 shows accumulation done in a nested loop. If the inner-loop is not unrolled, loop-iterations are launched every cycle. If the inner-loop is unrolled, you will get several parallel additions. Figure 4.21 shows the compilation report for Listing 4.27. Because of the parallel additions, loop-iterations are launched every 67 clock cycles.

```
__kernel void accumulation( __global const float * restrict din,
                __global float * restrict dout,
                int N )
{
  float acc = 0.0f;
  for(int i=0; i<N; i++)
  {
    #pragma unroll
    for(int j=0; j<16; j++)
    {
      acc += din[i*16+j];
```

```
Loop Report:

 + Loop "Block1" (file accsimple.cl line 6)
   Pipelined with successive iterations launched every 8 cycles due to:

       Data dependency on variable acc  (file accsimple.cl line 8)
       Largest Critical Path Contributor:
           96%: Fadd Operation  (file accsimple.cl line 8)
```

(a)

```
Loop Report:

 + Loop "Block2" (file accsimple.cl line 6)
   Pipelined well. Successive iterations are launched every cycle.
```

(b)

Fig. 4.20 Compilation report for Listing 4.26. (**a**) Compiled for Stratix V FPGAs. (**b**) Compiled for Arria 10 FPGAs

```
Loop Report:

 + Loop "Block2" (file accloopunroll.cl line 6)
 | Pipelined with successive iterations launched every 67 cycles due to:
 |
 |     Data dependency on variable
 |     Largest Critical Path Contributors:
 |         6%: Hardened Floating-Point Add Operation  (file accloopunroll.cl line 11)
 |         6%: Hardened Floating-Point Add Operation  (file accloopunroll.cl line 11)
 |         6%: Hardened Floating-Point Add Operation  (file accloopunroll.cl line 11)
 |         6%: Hardened Floating-Point Add Operation  (file accloopunroll.cl line 11)
 |         6%: Hardened Floating-Point Add Operation  (file accloopunroll.cl line 11)
 |         6%: Hardened Floating-Point Add Operation  (file accloopunroll.cl line 11)
 |         6%: Hardened Floating-Point Add Operation  (file accloopunroll.cl line 11)
 |         6%: Hardened Floating-Point Add Operation  (file accloopunroll.cl line 11)
 |         6%: Hardened Floating-Point Add Operation  (file accloopunroll.cl line 11)
 |         6%: Hardened Floating-Point Add Operation  (file accloopunroll.cl line 11)
 |         6%: Hardened Floating-Point Add Operation  (file accloopunroll.cl line 11)
 |         6%: Hardened Floating-Point Add Operation  (file accloopunroll.cl line 11)
 |         6%: Hardened Floating-Point Add Operation  (file accloopunroll.cl line 11)
 |         6%: Hardened Floating-Point Add Operation  (file accloopunroll.cl line 11)
 |
 |
 |-+ Fully unrolled loop (file accloopunroll.cl line 9)
    Loop was fully unrolled due to "#pragma unroll" annotation.
```

Fig. 4.21 Compilation report for Listing 4.27

```
    }
  }
  *dout = acc;
}
```

Listing 4.27 Floating-point accumulation in nested-loops

There are two methods to solve this problem for Arria 10 FPGAs. One method is to use the compilation flag `--fp-relaxed`. The other method is to modify the code according to Listing 4.28. These methods allow loop-iterations to be launched every clock cycle.

```
__kernel void accumulation(   __global const float * restrict din,
              __global float * restrict dout,
              int N )
{
  float acc = 0.0f;
  for(int i=0; i<N; i++)
  {
    float temp = 0.0f;
    #pragma unroll 16
    for(int j=0; j<16; j++)
    {
      temp += din[i*16+j];
    }
    acc += temp;
  }
  *dout = acc;
}
```

Listing 4.28 Kernel code optimized to utilized accumulators in Arria 10 FPGAs

References

1. Intel FPGA SDK for OpenCL, Programming guide (2017). https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf
2. Intel FPGA SDK for OpenCL, Best practices guide (2017). https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf
3. S.O. Settle, High-performance dynamic programming on FPGAs with OpenCL, in *IEEE High Performance Extreme Computing Conference (HPEC)* (2013), pp. 1–6
4. Z. Wang, B. He, W. Zhang, A study of data partitioning on OpenCL-based FPGAs, in *Field Programmable Logic and Applications (FPL)* (2015), pp. 1–8
5. E. Rucci, C. García, G. Botella, A.D. Giusti, M. Naiouf, M. Prieto-Matias, Smith-Waterman Protein Search with OpenCL on an FPGA, in *IEEE Trustcom/BigDataSE/ISPA* (2015), pp. 208–213
6. H.M. Waidyasooriya, Y. Takei, S. Tatsumi, M. Hariyama, OpenCL-Based FPGA-platform for stencil computation and its optimization methodology. IEEE Trans. Parallel Distrib. Syst. **28**(5), 1390–1402 (2017)
7. H.M. Waidyasooriya, M. Hariyama, K. Kasahara, OpenCL-based implementation of an FPGA accelerator for molecular dynamics simulation. Inf. Eng. Express Int. Inst. Appl. Inf. **3**(2), 11–23 (2017)
8. D.R. Kaeli, P. Mistry, D. Schaa, D.P. Zhang, *Heterogeneous Computing with OpenCL 2.0* (Morgan Kaufmann, San Francisco, 2015)
9. S. Tatsumi, M. Hariyama, M. Miura, K. Ito, T. Aoki, OpenCL-based design of an FPGA accelerator for phase-based correspondence matching, in *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)* (2015), pp. 613–617
10. Terasic, DE5-Net FPGA development kit (2012). http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=158&No=526
11. T. Endo, H.M. Waidyasooriya, M. Hariyama, Automatic optimization of OpenCL-based stencil codes for FPGAs, in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing* (Springer International Publishing, Berlin, 2017), pp.75–89
12. Terasic, DE5a-Net Arria 10 FPGA development kit (2017). https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=228&No=970&PartNo=2

Chapter 5
Exploiting the Memory Hierarchy

5.1 Overview of the Memory Hierarchy

The OpenCL for FPGA [1] uses a hierarchical memory structure [2, 3] as explained in Sect. 1.2.4. Table 5.1 shows how the different memories are implemented. The host memory is located in the host. The global memory is a DRAM located on the FPGA board outside the FPGA chip. The constant memory occupies a portion of the DRAM that is used as the global memory. The data of the constant memory is cached on the on-chip memory of the FPGA. Both the local and the private memories are on the FPGA chip [4, 5]. The local memory is implemented using RAM blocks. The private memory is implemented using both RAM blocks and registers.

5.2 Host Memory

The host memory is used in two ways. One way is to store the data for the computation in the host. The other way is to store the data that are transferred to the device and to store the results that are transferred from the device. For the latter way, the host memory should be allocated 64-byte aligned. This enables direct memory access (DMA) transfers between the host and the device. The following code shows how to align the host memory using the alignment size of 64-byte.

```
#define AOCL_ALIGNMENT 64

void *data_host = NULL;
posix_memalign (&data_host, AOCL_ALIGNMENT, size);
```


H.M. Waidyasooriya et al., *Design of FPGA-Based Computing Systems with OpenCL*, https://doi.org/10.1007/978-3-319-68161-0_5

Table 5.1 Implementations of different types of memories in OpenCL for FPGA

Memory type	Access	Implementation
Host memory	Host only: read/write	Host DRAM
Global memory	Host and all work items: read/write	FPGA DRAM (DDR3, DDR4, QDR)
Constant memory	Host: read write, all work items: read only	FPGA DRAM and RAM blocks
Local memory	Work group only: read/write	RAM blocks
Private memory	Work-item only: read/write	RAM blocks and registers

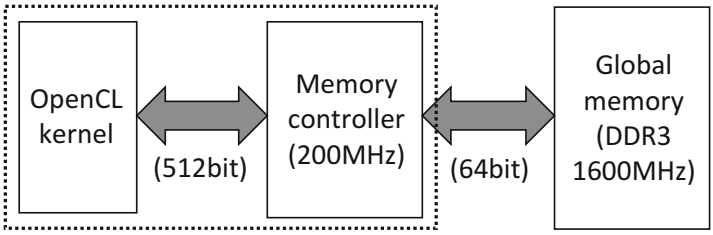


Fig. 5.1 Datapath between the global memory and a kernel

5.3 Global Memory

Figure 5.1 shows the datapath between the global memory and a kernel. The kernel accesses the global memory through a memory controller. A memory controller provides a simple interface to the kernel while taking care of the complicated control of the global memory. For example, the DDR3 memory used in Terasic DE5-Net FPGA Development Kit [6] operates at a high clock frequency (1600 MHz) and has a 64-bit data bus. The memory controller operates at a low clock frequency (200 MHz) and has a wider (512-bit) data bus.

A transaction refers to one read or write access to or from the memory. One transaction to global memory gives the access to one contiguous region of fixed size determined by the bus width between the kernel and the memory controller. In this example, 64 bytes (512 bits) are accessed in one transaction. Figure 5.2 shows that data $d1$ and $d2$ of 4 bytes are stored in distant 64-byte regions. To access $d1$ and $d2$, we need two transactions, and each transaction loads 64 bytes even if $d1$ and $d2$ are only 4 bytes.

If the data are located in distant 64-byte regions, multiple transactions are required as shown in Fig. 5.2. Merging multiple transactions into one is called "memory access coalescing" [7]. Coalescing is required for efficient memory access. In addition, coalescing simplifies the datapath of the memory access. This could reduce the resource utilization and increase the clock frequency.

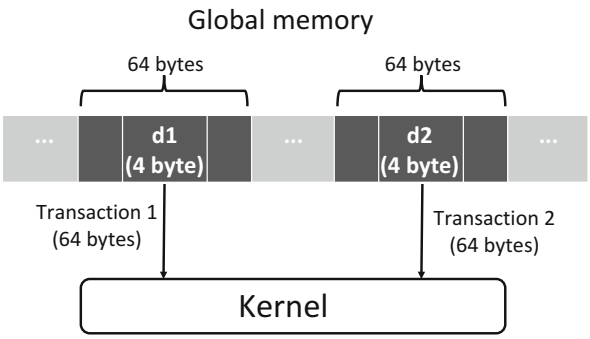


Fig. 5.2 Accessing data in distant 64-byte regions

5.3.1 Using Array-of-Structures for Coalescing

Built-in vector types such as `float2` and `float4` [8] can be used for the coalesced memory access. Custom data types are required to merge more complicated access patterns. Structure types can be used to merge several transactions into one. For example, the kernel in Listing 5.1 requires eight inputs per loop-iteration, and this leads to non-coalesced memory access. Listing 5.2 shows how to use a structure type to combine all the inputs into one, so as to coalesce the memory accesses.

```
#define N (1024 * 1024 * 32)

__kernel void myKernel( __global double * restrict A,
                        __global double * restrict B,
                        __global double * restrict C,
                        __global double * restrict D,
                        __global double * restrict E,
                        __global double * restrict F,
                        __global double * restrict G,
                        __global double * restrict H,
                        __global double * dout  )
{
  for(int i=0; i<N; i++)
  {
    dout[i] = A[i] + B[i] + C[i] + D[i] + E[i] + F[i] + G[i] + H[i];
  }
}
```

Listing 5.1 Non-coalesced memory access

```
#define N (1024 * 1024 * 32)

struct myStruct
{
  double A;
  double B;
  double C;
  double D;
  double E;
  double F;
  double G;
  double H;
}

__kernel void myKernel( __global struct myStruct * restrict din,
```

Table 5.2 Processing time of coalesced and non-coalesced access

Method	Processing time (ms)	Clock frequency (MHz)
Listing 5.1 (non-coalesced)	289	281.6
Listing 5.2 (coalesced)	233	304.6

```
                          __global double * restrict dout  )
{
  for(int i=0; i<N; i++)
  {
    dout[i] = din[i].A + din[i].B + din[i].C + din[i].D \
            + din[i].E + din[i].F + din[i].G + din[i].H;
  }
}
```

Listing 5.2 Using a structure type for coalescing

Table 5.2 shows the comparison of the processing times for non-coalesced access in Listing 5.1 and coalesced access in Listing 5.2. The evaluation is done for 33,554,432 loop-iterations and the FPGA board is the Terasic DE5-Net FPGA Development Kit [6]. The processing time for coalesced access is smaller than that for non-coalesced access. The clock frequency for coalesced access is higher because of the simpler datapath. Note that do not use a structure-of-arrays for coalescing. In this case, arrays are allocated to distant locations, so that many transactions occur.

5.3.1.1 How to Align Structures

Structure elements must be correctly aligned for improving performance. The offline compiler aligns structure elements according to the following criteria [4, 5] by default.

- The alignment must be a power of two.
- The alignment must be a multiple of the least-common-multiple word-width of the structure member sizes.

Let us consider the example in Listing 5.3. The kernel `myKernel` accesses an array of type `myStruct`. The members of `myStruct` are 1-byte wide "`char` type" and 4-byte wide "`int` type." Since the least-common-multiple word-width is 4 bytes and 4 is power of 2, the structure is 4-byte aligned as shown in Fig. 5.3. Since `char` is only 1-byte wide, three bytes of padding is inserted to achieve the 4-byte alignment. This increases the size of a structure element to 12 bytes instead of its original size of 9 bytes. Moreover, some of the structure elements belong to multiple 64-byte regions. Since multiple transactions are required to access those data, the structure elements are not properly aligned.

```
struct myStruct
{
  char A;   \\ 1 byte
  int B;    \\ 4 bytes
```

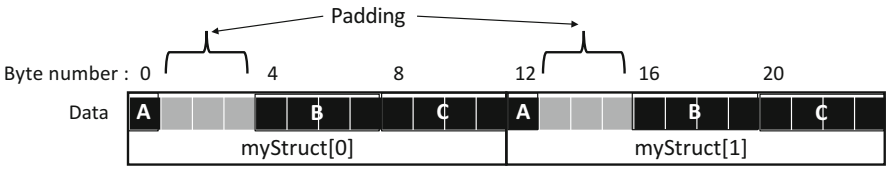


Fig. 5.3 Alignment of `myStruct` in Listing 5.3

Source: struct_pad...	Attributes	Stall%	Occupancy%	Bandwidth
int a = (int)din[i].A;	(__global{MEMORY},read)	(0.01%)	(77.7%)	(2340.3MB/s, 8.34%Efficiency)
int b = din[i].B;	(__global{MEMORY},read)	(0.01%)	(77.7%)	(2340.3MB/s, 66.67%Efficiency)
int c = din[i].C;				
dout[i] = a + b + c;	(__global{MEMORY},write)	(0.01%)	(77.7%)	(838.4MB/s, 93.05%Efficiency)

Fig. 5.4 Profiling results for Listing 5.3

```
  int C;     \\ 4 bytes
}

__kernel void myKernel( __global struct myStruct * restrict din,
                        __global float * restrict dout  )
{
  for(int i=0; i<N; i++)
  {
    dout[i] = (int)din[i].A + din[i].B + din[i].C;
  }
}
```

Listing 5.3 Implementation of a structure with padding

Figure 5.4 shows the profiling results for Listing 5.3. The efficiency of accessing `myStruct` member `A` is only 8.34%. The efficiency is reduced due to the insertion of padding between the array fields. Due to the inefficient memory access, the occupancy also reduced to only 77%.

5.3.1.2 Removing Padding

Padding can be removed by using `__attribute__ ((packed))` as shown in Listing 5.4. Since padding between the structure members are removed, all members are tightly packed as shown in Fig. 5.5. As a result, the size of a structure element is 9 bytes. Although the size is reduced, some of the structure elements belong to multiple 64-byte regions. Therefore, the structure is not properly aligned.

```
struct __atribute__ ((packed)) myStruct
{
  char A;    \\1 byte
  int B;     \\ 4 bytes
  int C;     \\ 4 bytes
}
```

Listing 5.4 Implementation of a structure without using padding

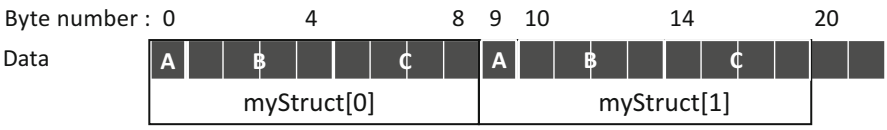


Fig. 5.5 Alignment of `myStruct` in Listing 5.4

Table 5.3 Processing time after removing padding

Method	Processing time (ms)
Listing 5.3 (with padding and misaligned)	84
Listing 5.4 (no padding and misaligned)	76

Source: struct_no_p...	Attributes	Stall%	Occupancy%	Bandwidth

int a = (int)din[i].A;
int b = din[i].B;
int c = din[i].C;
	(__global{MEMORY},read)	(0.01%)	(85.7%)	(1951.6MB/s, 100.00%Efficiency)

dout[i] = a + b + c;	(__global{MEMORY},write)	(0.01%)	(85.7%)	(874.7MB/s, 99.15%Efficiency)

Fig. 5.6 Profile results for Listing 5.4

The processing time using a non-padded structure is 9% smaller than that using a padded structure as shown in Table 5.3. Figure 5.6 shows the profiling results after the padding is removed. The efficiency is nearly 100%. Although the occupancy is also improved to 83.7%, the structure is still misaligned since some of the structure elements belong to multiple 64-byte regions.

5.3.1.3 Aligning and Removing Padding

To improve the performance, we have to properly align the structure elements so that any structure element belongs to a single transaction region of 64 bytes. For this purpose, the transaction size must be a multiple of the alignment size. Listing 5.5 shows the kernel code to properly align the structure elements by using `__attribute__ ((packed))` and `__attribute__ ((aligned(N)))` together, where `N` is the alignment size in bytes. Since the transaction size of 64 bytes is a multiple of alignment size of 16 bytes, each structure element belongs to a single 64-byte region as shown in Fig. 5.7. This provides the most efficient hardware and efficient memory access.

```
struct __atribute__ ((packed)) __atribute__ ((aligned(16))) myStruct
{
  char A;    \\1 byte
  int B;     \\ 4 bytes
  int C;     \\ 4 bytes
}
```

Listing 5.5 Alignment using the attributes `packed` and `aligned` together

Table 5.4 Comparison of the processing times for different alignment methods

Method	Processing time (ms)
Listing 5.3 (with padding and misaligned)	84
Listing 5.4 (no padding and misaligned)	76
Listing 5.5 (no padding and aligned)	60

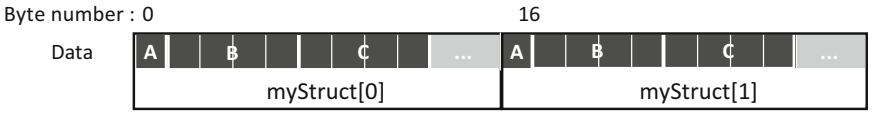


Fig. 5.7 Alignment using the attributes `packed` and `aligned` together

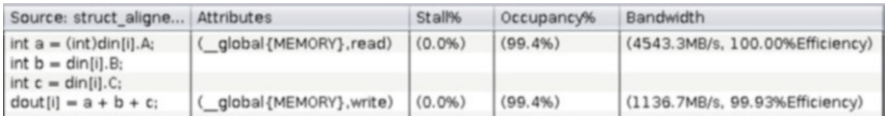


Fig. 5.8 Profile results when using a properly aligned structure

Table 5.4 shows the comparison of the processing times for different alignment methods. The processing time using an aligned structure is 21% smaller compared to the misaligned structure without padding. Figure 5.8 shows the profiling results. The efficiency is 100% and the occupancy is almost 100%.

5.3.2 Interleaving vs. Non-interleaving

Many FPGA boards contain multiple memory modules that can be accessed in parallel. They are called "memory banks." By default, data are evenly allocated onto multiple banks to avoid concentration of memory access on one particular bank. Figure 5.9 shows the interleaved and non-interleaved memory access. In non-interleaved memory access, the top half of the addresses is used by bank 1, while the bottom half is used by bank 2, as shown in Fig. 5.9a. In the interleaved memory, the addresses for memory banks are located alternatively as shown in Fig. 5.9b. The address space of the first 1024 bytes belongs to bank 1 and the next 1024 bytes belongs to bank 2. This is repeated for every 1024 bytes. Therefore, the access of a large data array is distributed evenly to both memory banks.

Interleaved memory access is the default configuration in OpenCL for FPGA. This is very convenient since the programmer does not have to consider the number of banks when writing the code. The memory interleaving can be disabled using the compiler option `--no-interleaving default`. Non-interleaved memory access provides a simple datapath between the kernel and memory. Therefore, a kernel with non-interleaved memory access consumes less resources compared to a kernel with interleaved memory access. When the memory interleaving is disabled, the programmer has to manually assign memory objects to banks as follows.

Address	Bank number
0x0000 ~ 0x03FF	bank 1
0x0400 ~ 0x07FF	bank 1
...	...
0x1000 ~ 0x13FF	bank 2
0x1400 ~ 0x17FF	bank 2
...	...

(a)

Address	Bank number
0x0000 ~ 0x03FF	bank 1
0x0400 ~ 0x07FF	bank 2
...	...
0x1000 ~ 0x13FF	bank 1
0x1400 ~ 0x17FF	bank 2
...	...

(b)

Fig. 5.9 Interleaved and non-interleaved memory access. (**a**) Non-interleaved memory. (**b**) Interleaved memory

```
main ()
{
  mobj_din  = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_BANK_1_ALTERA,
     sizeof(float)*SIZE, NULL, &status);
  mobj_dout = clCreateBuffer(context, CL_MEM_WRITE_ONLY | CL_MEM_BANK_2_ALTERA,
      sizeof(float)*SIZE, NULL, &status);
}
```

The flags `CL_MEM_BANK_1_ALTERA` and `CL_MEM_BANK_2_ALTERA` represent the banks 1 and 2, respectively. Note that, in SDK for OpenCL version 17.0 and above, the names of the flags are changed to `CL_MEM_BANK_1_INTEL` and `CL_MEM_BANK_2_INTEL`.

Non-interleaved memory access is usually used when there are read-only and write-only data. In this case, one memory bank is used for the read-only data while the other is used for the write-only data. This eliminates frequent read and write accesses to the same bank and increases the memory access bandwidth. For the other cases, whether to use interleaved memory access or non-interleaved memory access is a difficult question. To explain this, let us consider the example shown in Listing 5.6. Let us consider array `A` of type `TypeA` and array `B` of type `TypeB` which are stored in the global memory. The number of bytes required in one clock cycle is determined by the sizes of `TypeA` and `TypeB`. We consider four different configurations as shown in Table 5.5. The required bandwidth is given by

$$(sizeof(\text{TypeA}) + sizeof(\text{TypeB})) \times N \times \text{frequency}. \tag{5.1}$$

```
__kernel void myKernel ( __global TypeA restrict * A,
                         __global TypeB restrict * B,
                         ...  )
{
  for(int i=0; i<N; i++)
  {
    //Some computation using A and B
    dout[i] = A[i] operation B[i];
  }
}
```

Listing 5.6 Example of global memory access using two arrays of `TypeA` and `TypeB`

Table 5.5 Required bandwidth for different memory configurations

Configuration	TypeA	TypeB	Required bandwidth (GB/s)
1	int8 (32 bytes)	int (4 bytes)	8.64
2	int8 (32 bytes)	int8 (32 bytes)	15.36
3	int16 (64 bytes)	int (4 bytes)	16.32
4	int16 (64 bytes)	int16 (64 bytes)	30.72

To evaluate the performance of interleaved and non-interleaved memory access, experiments are conducted using Terasic DE5-Net FPGA Development Kit [6]. It has two memory banks where each has a theoretical bandwidth of 12.8 GB/s. The clock frequency is 240 MHz. For interleaved access, all the default settings are used. For non-interleaved access, arrays `A` and `B` are allocated to bank 1 and bank 2, respectively. Figure 5.10 shows the bandwidth of each configuration. In configuration 1, the size of array `A` is 8 times larger than that of array `B`. For non-interleaved access, the bandwidth of bank 1 is 8 times larger than that of bank 2. For interleaved access, the bandwidth of both banks are equal to each other since data are evenly allocated to both banks. In configuration 2, the sizes of both arrays are the same, so that interleaved and non-interleaved accesses give the similar results. In configuration 3, the bandwidth of interleaved access is larger than that of non-interleaved access. Interleaved access can facilitate the total required bandwidth of 16.32 GB/s by utilizing both banks equally. In non-interleaved access, the required bandwidth from bank A and bank B are 15.36 GB/s and 0.96 GB/s, respectively. Bank A cannot facilitate the required bandwidth 15.36 GB/s, since it exceeds the theoretical bandwidth (12.8 GB/s). Therefore, a memory access bottleneck exists in Bank 1, so that the total bandwidth is restricted. However, bank 2 is under-utilized since the required bandwidth for array `B` is only 0.96 GB/s. In configuration 4, non-interleaved access is much better compared to interleaved access. When the cases where the required bandwidth is close to the theoretical bandwidth, non-interleaved access performs better than interleaved access because of its simple datapath.

We summarize how to use interleaved and non-interleaved access as follows. If the required bandwidths of all memory banks are much smaller than the theoretical bandwidth, both interleaved and non-interleaved access give similar performances. If the required bandwidth of one bank is significantly different from that of another banks, and if there is a memory access bottleneck in at least one bank, interleaved memory access is the better option. If there are memory access bottlenecks in all banks, non-interleaved memory access is better. It has a simple hardware that is capable of providing a bandwidth close to the theoretical one. In all other situations, it is difficult to predict which memory configuration is better. Therefore, you had better compile the kernel for both configurations and choose the better one. Note that, it is impossible to achieve the theoretical memory bandwidth, and achieving around 90% of the theoretical bandwidth can be regarded as extremely good.

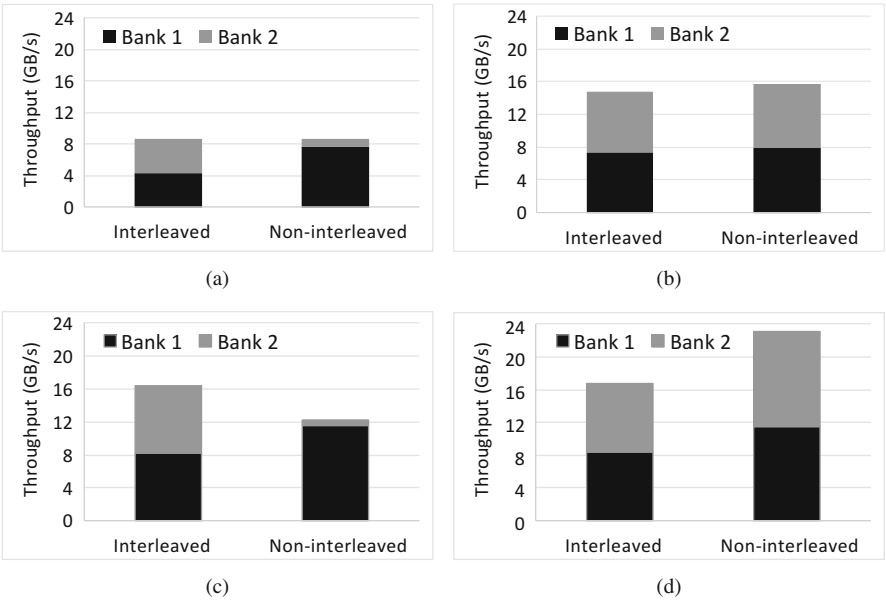


Fig. 5.10 Memory access bandwidths for different configurations. (**a**) Imbalanced access. Small data size. (**b**) Balanced access. Small data size (**c**) Imbalanced access. Large data size (**d**) Balanced access. Large data size

5.3.3 *Using Different Global Memory Types*

Depending on the FPGA board and its BSP, there can be different global memory types such as `DDR` or `QDR`. Usually, the DDR memory is suitable for long sequential access while low latency QDR memory is suitable for random access. The types and names of the global memory can be found in board vendor's documentation or in the *board_spec.xml* file. The memory type can be specified in a kernel by using `buffer_location` attribute. The code in Listing 5.7 shows how the memory type is specified. A pre-processor macro is defined for each memory type, so that you can use the macro name in the kernel. Host allocate data to the specified memory type. If the memory type is not mentioned in the kernel code, host allocate data to the default memory type. The default memory type is one defined first, or one that contains `default=1` flag, in the *board_spec.xml* file.

```
#define QDR __global __attribute__((buffer_location("QDR")))
#define DDR __global __attribute__((buffer_location("DDR")))

__kernel void myKernel( QDR uint * A, DDR uint * B )
{
    //statements
}
```

Listing 5.7 The use of different global memory types

5.4 Constant Memory

The constant memory is a read-only memory that resides in global memory. The data in the constant memory are automatically loaded into the on-chip cache at runtime. The cache is implemented using on-chip RAM blocks. The default size of the constant memory is 16 kB. You can specify the constant memory size by including the compiler option `--const-cache-bytes <N>`, where `<N>` is the constant memory size in bytes. The following code shows how to use the constant memory in a kernel.

```
__kernel void myKernel ( __constant int * A, ... )
{
  ...
}
```

Qualifier `__constant` defines the constant memory arrays. Since constant memory suffers large performance penalties for cache misses, it is suitable for the data that are large enough to be placed in the on-chip memory of an FPGA. If the constant memory size does not fit in the cache, you can use the global memory declared with `__global const` qualifier. This simplifies the global memory access of the kernel. Note that the already-cached data are invalidated when the host writes to the constant memory.

If the host always passes the same constants to the kernel, you can use those data directly in the FPGA kernel (without transferring from the host) by declaring the constant variables in file scope. This is shown in Listing 5.8. This generates a ROM (read-only-memory) directly in the FPGA to store the constants. Note that you cannot declare vector type constants (or arrays) inside a kernel function in OpenCL for FPGA [4, 5].

```
__constant int A[4] = {1,2,6,8};
__kernel void myKernel ( ... , ... )
{
  ...
}
```

Listing 5.8 Kernel code to declare constant memory in file scope

5.5 Local Memory

The local memory is declared using `__local` or `local` qualifier. The Local memory can have one or more banks. A bank has two ports that can be accessed simultaneously. By default, one port is used for the write access and the other is used for the read access. There are several ways to increase the parallel data access of the local memory.

- Increase the number of banks.
- Use double pumping to double the number of virtual ports.
- Increase the number of ports by using local memory replication.

5.5.1 Local Memory Banks

The data stored in separate banks are accessed in parallel. The number of banks of a local memory can be defined manually, so as to increase the parallel access. The attributes `numbnks()` and `bankwidth()` are used to configure the number of banks and the data-bus width, respectively. Figure 5.11 shows different configurations of the local memory banks.

5.5.2 Double Pumping

In OpenCL for FPGA, a method called double pumping is used to increase the number of ports of the local memory. In this method, the local-memory-clock frequency is set to the double of the kernel-clock frequency. As a result, local memory is accessed twice in one kernel-clock cycle, which could also be seen as doubling the number of ports. Figure 5.12 shows the implementation of double pumping. Although the RAM block has only two ports, we can access four memory address from four virtual ports in one kernel-clock cycle.

As explained above, the local-memory-clock frequency is set to the double of the kernel-clock frequency for double pumping. For large kernels, the offline compiler may not be able to implement a high clock frequency for the local memory. Therefore, the offline compiler reduces the kernel-clock frequency.

Advantages of using double pumping

- Increases from one read port to three read ports
- Saves RAM usage

Disadvantages of using double pumping:

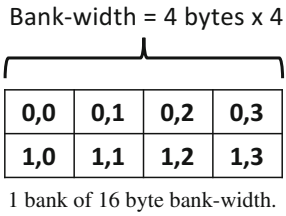
- Implements additional logic
- Might reduce maximum frequency

The attribute `singlepump` or `doublepump` can be used to specify whether to disable or enable the double pumping feature, respectively. Listing 5.9 shows how to specify the double pumping. If you want to specify the single pumping, replace `doublepump` with `singlepump` in Listing 5.9.

```
int __attribute__((memory,
                    doublepump,
                    ...
                   ))
                   lmem[16];
```

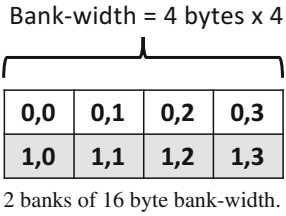
Listing 5.9 Implementation of local memory with double pumping

```
local int __attribute__((numbanks(1),
                         bankwidth(16)))
                         lmem[2][4];
```



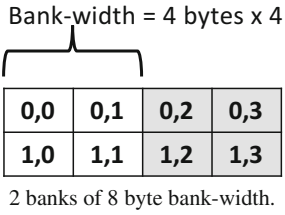
1 bank of 16 byte bank-width.

```
local int __attribute__((numbanks(2),
                         bankwidth(16)))
                         lmem[2][4];
```



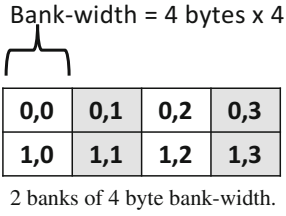
2 banks of 16 byte bank-width.

```
local int __attribute__((numbanks(2),
                         bankwidth(8)))
                         lmem[2][4];
```



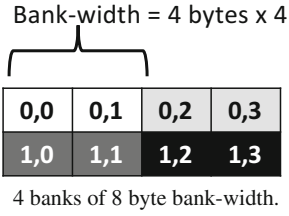
2 banks of 8 byte bank-width.

```
local int __attribute__((numbanks(2),
                         bankwidth(4)))
                         lmem[2][4];
```



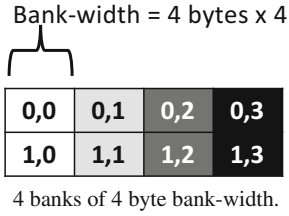
2 banks of 4 byte bank-width.

```
local int __attribute__((numbanks(4),
                         bankwidth(8)))
                         lmem[2][4];
```



4 banks of 8 byte bank-width.

```
local int __attribute__((numbanks(4),
                         bankwidth(4)))
                         lmem[2][4];
```



4 banks of 4 byte bank-width.

Fig. 5.11 Local memory configurations

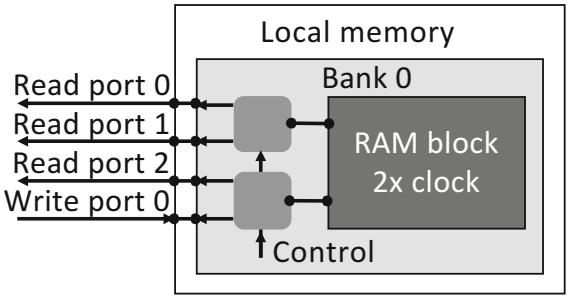


Fig. 5.12 Double pumping

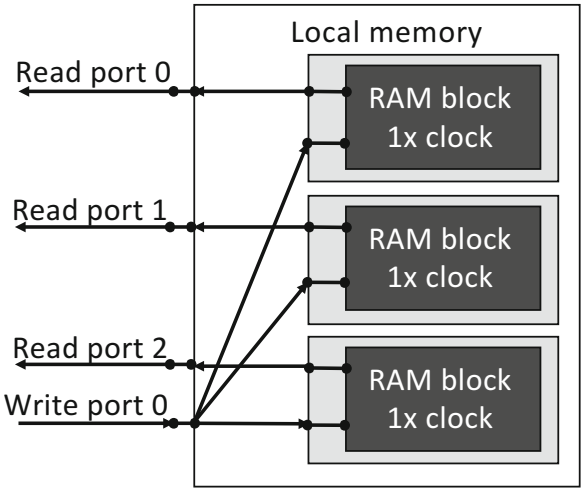


Fig. 5.13 Local memory created by replication

5.5.3 *Local Memory Replication*

In order to create multiple ports, the offline compiler can replicate the local memory using multiple RAM blocks [4, 5]. Attributes `numreadports()` and `numwriteports()` are used to specify the number of ports. The example in Listing 5.10 shows a local memory that has 3 read ports and 1 write port. Figure 5.13 shows the local memory created by replicating RAM blocks. All three RAM blocks have the same data. The write port of all three RAM blocks are connected, so that all three RAM blocks are updated simultaneously. The read ports of all RAM blocks are accessed in parallel. As a result, parallel access to all ports is possible.

```
int __attribute__((memory,
                   singlepump,
                   numreadports(3),
                   numwriteports(1)))
                   lmem[16];
```

Listing 5.10 Kernel code for local memory replication

5.6 Private Memory

The private memory is the on-chip memory dedicated to a single work-item. The variables inside a kernel are implemented as the private memory if they do not have any qualifier. The private memory is implemented by either registers or RAM blocks, depending on the memory size and the kernel type.

Shift-registers are very important in OpenCL-based accelerator design. Listing 5.11 shows the source code of the shift-register implementation. Shift-registers are declared in a similar way to an array. The size of the shift-register must be fixed. The initialization of shift-registers is shown from lines 8–12. If the initial values are not required, you can skip this step. The shift-register is generated by unrolling the loop as shown from lines 16–20. Line 21 shows how the data is pushed into the shift-register. Note that it is possible to access multiple point of the shift-register array in parallel.

```
1  #define SIZE 1024   //the shift-register size must be a constant
2  __kernel void myKernel ( ... )
3  {
4    int shift_reg[SIZE];
5
6    //initialize shift-registers
7    //All elements of the array should be initialized to the same value
8    #pragma unroll
9    for (int i=0; i < SIZE; i++)
10   {
11     shift_reg[i] = 0;
12   }
13
14   #pragma unroll
15   for (int j=SIZE-1; j>0; j++)
16   {
17     shift_reg[j] = shift_reg[j-1];
18   }
19   shift_reg[0] = ...;     //push data into the shift-register
20   ...
21
22 }
```

Listing 5.11 Implementing a shift-register

5.7 Channels

There are two methods to transfer data between kernels as shown in Fig. 5.14. One method is to use global memory similar to GPUs. This method is shown in Fig. 5.14a. It requires accesses to the global memory and this could cause memory access bottlenecks.

The other method is to use channels. Figure 5.14b shows how to transfer data between kernels using channels. A dedicated data-path called channel is implemented between two kernels, using RAM blocks and registers. A channel is a FIFO (first-in first-out). The producer kernel can write data to a channel if it is

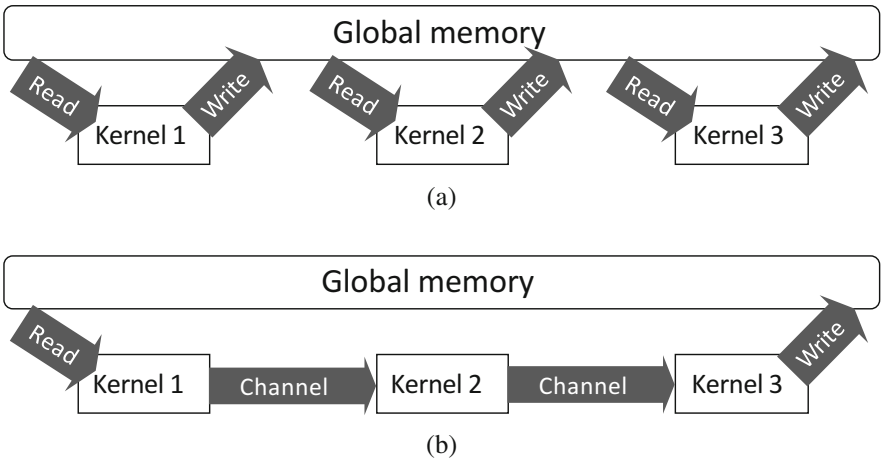


Fig. 5.14 Data transfers among kernels. (**a**) Data are transfer among kernels using global memory. (**b**) Data are transfer among kernels using channels

not fully filled. The consumer kernel reads the data from the channel. The use of channels allows the kernels to transfer data between each other without accessing the global memory. The use of channels is strongly recommended since the memory bandwidth of FPGAs is smaller than those of CPUs and GPUs.

The speed of writing data to a channel and the speed of reading data from the same channel could be different. As a result, one kernel has to wait until the other kernel finishes the channel operation. To avoid or minimize such imbalances of data transfers, you can add buffers to the channels. This is done by adding `depth` attribute as shown in Listing 5.12, where a FIFO of 8 stages are added to the channel. If the offline compiler can detect imbalances, it automatically adds buffers to a channel [4, 5].

```
channel int __attribute__((depth(8))) c0;

__kernel void myChannel(__global int * in_buf,)
{
  for (int i=0; i<100; i++)
    write_channel_altera(c0, in_buf[i]);
}
```

Listing 5.12 Specifying the depth of a channel

References

1. SDK for OpenCL (2017). https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html
2. J.E. Stone, D. Gohara, G. Shi, OpenCL: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66–73 (2010)

3. D.R. Kaeli, P. Mistry, D. Schaa, D.P. Zhang, *Heterogeneous Computing with OpenCL 2.0* (Morgan Kaufmann, Waltham, 2015)
4. Intel FPGA SDK for OpenCL, Programming Guide (2017). https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf
5. Intel FPGA SDK for OpenCL, Best Practices Guide (2017). https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf
6. Terasic, DE5-Net FPGA Development Kit (2012). http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=158&No=526
7. J.W. Davidson, S. Jinturkar, Memory access coalescing: a technique for eliminating redundant memory accesses, in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (1994), pp. 186–195
8. Khronos group, The open standard for parallel programming of heterogeneous systems (2017). https://www.khronos.org/opencl/

Chapter 6
Design Examples

6.1 Design of a Stencil-Computation Accelerator

Stencil computation is widely used in scientific computations such as fluid dynamics [1], simulations, electromagnetic simulations [2], iterative solvers [3], etc. In this section, we analyze the stencil computation to understand different types of parallelism and combine these parallelism optically. Then we discuss its architecture and what type of OpenCL kernel we should use for the implementation. We also discuss how to fine-tune the stencil computation architecture to achieve best performance for different FPGAs.

6.1.1 Stencil Computation

A stencil is a shape that consists of neighboring grid-points called cells. Figure 6.1 shows a 2-D 5-point stencil. The data of these five cells at the current iteration are used to compute a value of the center cell at the next iteration. The typical computation is a sum of products as given by Eq. (6.1), where the iteration number and grid coordinates are given by t and (x, y), respectively. This computation is repeated to compute values for all cells in iteration $t + 1$.

$$\begin{aligned} cell^{t+1}_{(x,y)} = {} & k_1 \times cell^{t}_{(x,y-1)} + k_2 \times cell^{t}_{(x-1,y)} + k_3 \times cell^{t}_{(x,y)} \\ & + k_4 \times cell^{t}_{(x+1,y)} + k_5 \times cell^{t}_{(x,y+1)}. \end{aligned} \tag{6.1}$$

Figure 6.2 shows the iterative computation. The data of iteration $t - 1$ are used to compute the data of iteration t. Similarly, the data of iteration t are used for the computation in iteration $t + 1$.

H.M. Waidyasooriya et al., *Design of FPGA-Based Computing Systems with OpenCL*, https://doi.org/10.1007/978-3-319-68161-0_6

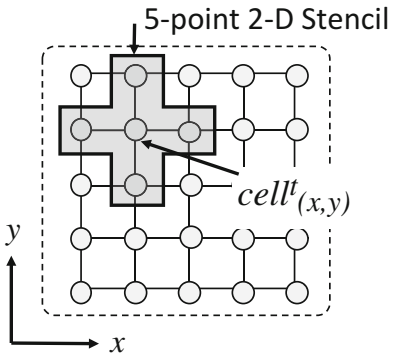


Fig. 6.1 Stencil computation using a 2-D 5-point stencil

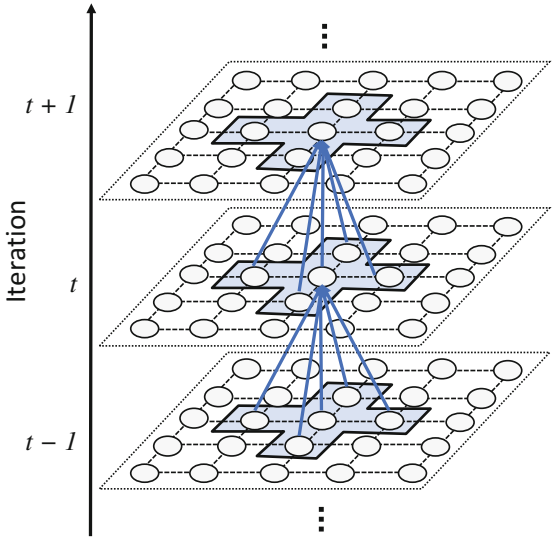


Fig. 6.2 Iterative computation

The CPU-based stencil computation source code is shown in Listing 6.1. The stencil computation of an iteration is done in `CPU_iteration` function according to Eq. (6.1). In the `main` function, the output of the current iteration is used as the input of the next iteration.

```
void CPU_iteration(float din, float dout) {
  for(int y=1; y<H-1; y++) {
    for(int x=1; x<W-1; x++) {
      // Compute one stencil according to Eq.(6.1)
      dout[y*W + x] = k1*din[(y-1)*W + x] +
                      k2*din[(y)*W + x-1] +
                      k3*din[(y)*W + x] +
                      k4*din[(y)*W + x+1] +
                      k5*din[(y+1)*W + x];
    }
  }
}

int main() {
  for(int i=0; i<ITERATION; x++)
```

```
  {
    if((i%2)==0) CPU_stencil(buf0, buf1);
    else         CPU_stencil(buf1, buf0);
  }
}
```

Listing 6.1 Stencil computation code for a CPU

There are two types of parallel computations available in stencil computation. One is the "stencil-parallel computation" and the other is the "iteration-parallel computation." Figure 6.3 shows the stencil-parallel computation, where the computations for the stencils centered at $(1, 1)$ and $(1, 2)$ in the same iteration t are done in parallel. Since the computations of the stencils in the same iteration are independent of each other, the computations can be done in any order and either in serial or in parallel. In order to compute multiple stencils in parallel, we have to access the data of those stencils in parallel too. Since the data are stored in the global memory, a huge global-memory bandwidth is required. Therefore, the processing speed is restricted by the global-memory bandwidth.

Figure 6.4 shows the iteration-parallel computation. Assumed that the computation of an iteration is done in a raster-scan manner from the bottom left. The gray region indicates that the values of the cells have been obtained. In other words, the

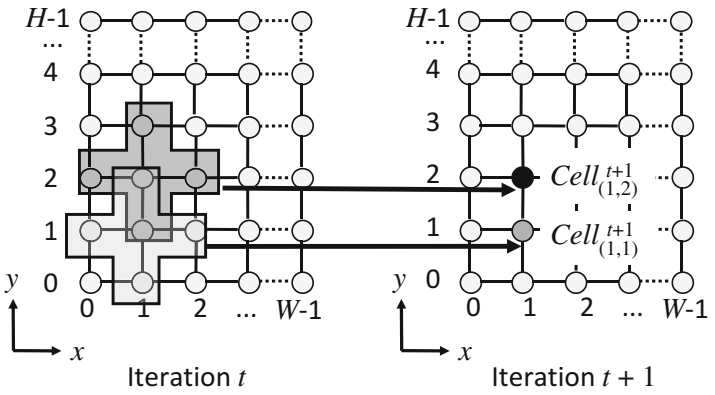


Fig. 6.3 Stencil-parallel computation

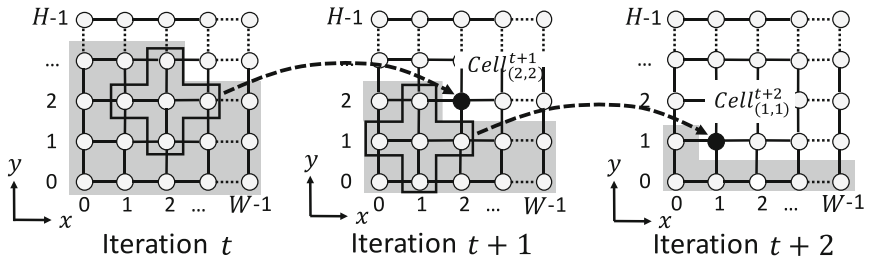


Fig. 6.4 Iteration-parallel computation

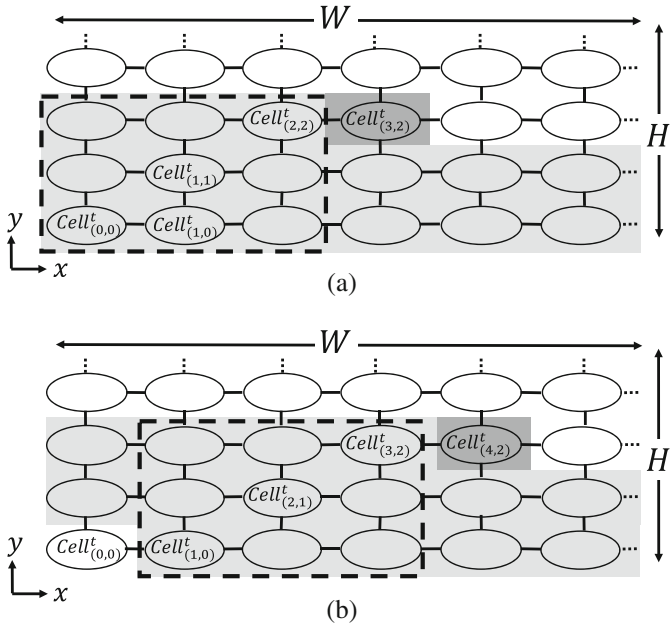


Fig. 6.5 Data access order in an iteration. (**a**) The computation for the stencil centered at $(1, 1)$ in iteration t is done after loading the value of $cell^t_{(2,2)}$. (**b**) The computation for the stencil centered at $(2, 1)$ in iteration t is done after loading the value of $cell^t_{(3,2)}$

values of the cells in the gray region are available. Computations of iteration $t + 1$ requires the data of iteration t. During the computation for the stencil centered at $(2, 2)$ in iteration t, the data needed for the computation for the stencils centered at $(1, 1)$ in iteration $t + 1$ are available. Therefore, the computations for the stencils centered at $(2, 2)$ in iteration t and $(1, 1)$ in iteration $t + 1$ can be done in parallel. We call this iteration-parallel computation.

The stencil-parallel computation is not suitable for FPGAs, since the memory bandwidths of FPGAs are smaller than those of CPUs and GPUs. Hence, many works such as [4–6] use iteration-parallel computation. This is because the results of iterations can be stored inside the FPGA without accessing the global memory by utilizing a large number of on-chip memories and registers.

Figure 6.5 shows the data access order in an iteration. For the simplicity, we consider a 3×3 stencil and a $W \times H$ gird. We assume that the cell data are loaded from the memory in the raster scan manner. The computation for the stencil centered at $(1, 1)$ in iteration t is done after loading the value of $cell^t_{(2,2)}$ as shown in Fig. 6.5a. The computation for the stencil centered at $(2, 1)$ in iteration t is done in the next clock cycle after loading the value of $cell^t_{(3,2)}$ as shown in Fig. 6.5b. At this time, the value of $cell^t_{(0,0)}$ is no longer required for any further computation. Similarly, the value of $cell^t_{(1,0)}$ will become obsolete in the next clock cycle. Therefore, a data value must be stored for a period of $2W + 4$ cycle at least. We call this period the lifetime of data.

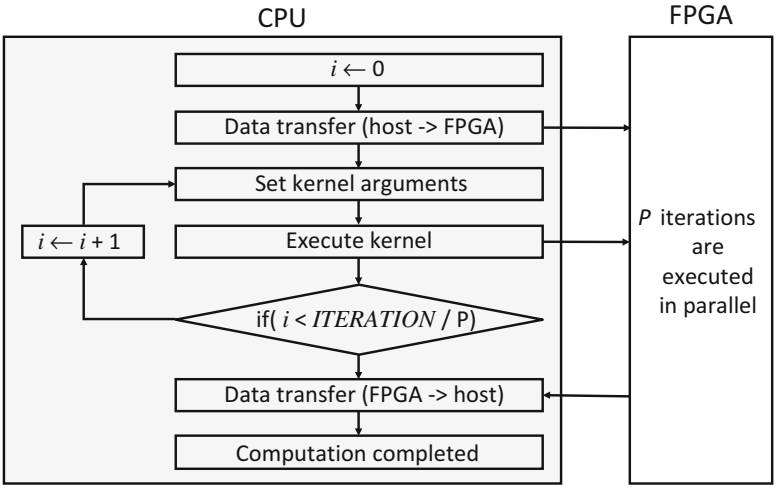


Fig. 6.6 Flowchart of the whole computation

The flowchart in Fig. 6.6 shows how the stencil computation is done. First, the input data are transferred from the CPU to the device (the FPGA board). Then the kernel arguments are set and the kernel is executed by the CPU. The kernel processes P iterations in the FPGA. Then, the output of the current kernel execution is set as the input of the next kernel execution by the CPU. The kernel is executed for $ITERATION/P$ times where $ITERATION$ denotes the number of iterations in the stencil computation. Finally, the computed results are transferred from the FPGA to the CPU.

6.1.2 OpenCL-Based Implementation

Figure 6.7 shows the FPGA architecture for stencil computation that is proposed in [7]. It consists of a DRAM and P "pipelined computation modules" (PCMs). The computation of an iteration is done in a PCM. A PCM consists of shift-registers and multiple processing elements (PEs). The computation of a stencil is done in a PE. Shift-registers are used to transfer the computed results of one PCM to the next PCM.

Figure 6.8 shows the shift register array required to implement the above-mentioned data-flow. Since the lifetime is $2W + 4$, a data value should be stored in the shift-registers for the same number of clock cycles. That means, we need a shift-register with $2W + 4$ in length. After the shift-registers are filled, the older data are pushed out while new data are pushed in.

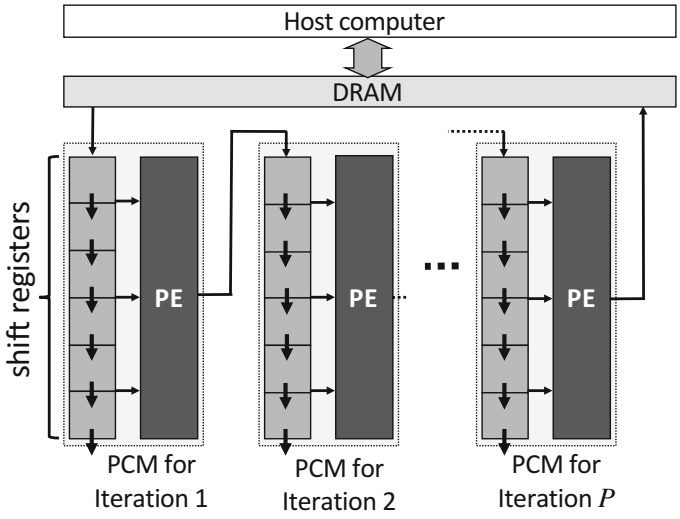


Fig. 6.7 Architecture proposed in [7] for stencil computation

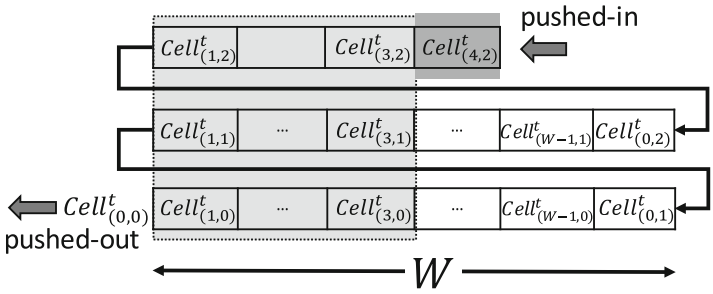


Fig. 6.8 Stencil computation using a 5-point 2-D stencil at iteration t. The stencil is moved over the 2-D grid to compute the values of the *cell*s in iteration t. The inputs for the computation are the data of iteration $t-1$

6.1.2.1 Kernel Code

In this section, we explain how to design the OpenCL kernel code to generate the accelerator shown in Fig. 6.7. Let us begin with declaring the kernel and specifying its inputs as the following code.

```
__attribute__((task))
__kernel void stencil( __global const float * restrict din,
                       __global float * restrict dout )
```

Task attribute (`task`) is used to declare a single work-item kernel. Since this kernel processes P iterations, the input data of the first iteration is `din` and the result of the iteration $(P-1)$ is `dout`. Arrays `din` and `dout` are read-only and write-only, respectively. For read-only data access, there are two optimization methods.

- Using constant cache
- Using qualifier `global const`

As explained in Sect. 5.4, for the case where the data size is larger than the cache size, the use of constant cache is not efficient. Our stencil computation belongs to the case. Therefore, we use qualifier `global const`. This allows the offline compiler to perform more aggressive optimization.

The following shows how to implement the shift-registers.

```
float shiftreg[P][2*W+4];

while (count != loop_iteration)
{
  #pragma unroll
  for (int i = 2*W+3; i>0; --i)
  {
    #pragma unroll
    for (int j=0; j<P; j++ )
    {
      shiftreg[j][i] = shiftreg[j][i-1];
    }
  }
  shiftreg[0][0] = (count < W*H) ? din[count] : 0.0f;
  ...
  ...
}
```

The length of the shift-registers equals the lifetime. The input data are written to the first shift register in iteration 0. In each loop-iteration, the data of the shift registers of all iterations move forward. The data in the last stage are discarded.

The computation of $P-1$ iterations (iterations 0 to $P-2$) is defined as follows.

```
float result;
float ce, le, ri, to, bo;

#pragma unroll
for (int j=0; j<P-1; j++)
{
  to = shiftreg[j][2*W+2];
  le = shiftreg[j][1*W+3];
  ce = shiftreg[j][1*W+2];
  ri = shiftreg[j][1*W+1];
  bo = shiftreg[j][0*W+2];

  if( count >= ((j+1)*(W+2)+W) && (((count-(j+1)*(W+2)) % W) != 0) &&
    ((count-(j+1)*(W+2)) % W) != (W-1) && count < ((j+1)*(W+2)+W*(H-1)) )
    result = k1*to + k2*le + k3*ce + k4*ri + k5*bo;
  else
    result = 0.0f;

  shiftreg[j+1][0] = result;
}
```

The `#pragma unroll` directive fully unroll the loop so that $P-1$ iterations are processed in parallel. The data required for the computation of a stencil are accessed from the shift-registers. We use a fixed value zero for the boundary data. In the last iteration (iteration $P-1$), the computation results are written to the global memory as follows.

```
int j = P-1;
to = shiftreg[j][2*W+2];
le = shiftreg[j][1*W+3];
ce = shiftreg[j][1*W+2];
ri = shiftreg[j][1*W+1];
bo = shiftreg[j][0*W+2];

if(count >= (j+1)*(W+2))
{
  if( count >= ((j+1)*(W+2)+W) && (((count-(j+1)*(W+2)) % W) != 0) &&
    ((count-(j+1)*(W+2)) % W) != (W-1) && count < ((j+1)*(W+2)+W*(H-1)) )
    result = k1*to + k2*le + k3*ce + k4*ri + k5*bo;
  else
    result = 0.0f;

  dout[count-(j+1)*(W+2)] = result;
}
```

The outputs of the final iteration are stored in the global memory instead of shift-registers.

The complete kernel code is given by Listing 6.2. As we can see, the code has only 61 lines. If we use HDL, the size of the code could be over a few thousand lines.

```
__attribute__((task))
__kernel void stencil( __global const float * restrict din,
                       __global float * restrict dout )
{
  float shiftreg[P][2*W+4];
  const int loop_iteration = P*(W+2) + W*H;
  int count = 0;

  while (count != loop_iteration)
  {
    #pragma unroll
    for (int i = 2*W+3; i>0; --i)
    {
      #pragma unroll
      for (int j=0; j<P; j++ )
      {
        shiftreg[j][i] = shiftreg[j][i-1];
      }
    }
    shiftreg[0][0] = (count < W*H) ? din[count] : 0.0f;

    float result;
    float ce, le, ri, to, bo;
    #pragma unroll
    for (int j=0; j<P-1; j++)
    {
      to = shiftreg[j][2*W+2];
      le = shiftreg[j][1*W+3];
      ce = shiftreg[j][1*W+2];
      ri = shiftreg[j][1*W+1];
      bo = shiftreg[j][0*W+2];

      if( count >= ((j+1)*(W+2)+W) && (((count-(j+1)*(W+2)) % W) != 0) &&
        ((count-(j+1)*(W+2)) % W) != (W-1) && count < ((j+1)*(W+2)+W*(H-1)) )
        result = k1*to + k2*le + k3*ce + k4*ri + k5*bo;
      else
        result = 0.0f;

      shiftreg[j+1][0] = result;
    }

```

```
    int j = P-1;
    to = shiftreg[j][2*W+2];
    le = shiftreg[j][1*W+3];
    ce = shiftreg[j][1*W+2];
    ri = shiftreg[j][1*W+1];
    bo = shiftreg[j][0*W+2];

    if(count >= (j+1)*(W+2))
    {
      if( count >= ((j+1)*(W+2)+W) && (((count-(j+1)*(W+2)) % W) != 0) &&
        ((count-(j+1)*(W+2)) % W) != (W-1) && count < ((j+1)*(W+2)+W*(H-1)) )
        result = k1*to + k2*le + k3*ce + k4*ri + k5*bo;
      else
        result = 0.0f;

      dout[count-(j+1)*(W+2)] = result;
    }
    count++;
  }
}
```

Listing 6.2 Kernel code for stencil computation

6.1.2.2 Host Code

In the host code, we use the output of the current iteration as the input of the next iteration. This is done by altering the order of the arguments of the kernel as shown in Listing 6.3.

```
for(int x=0; x<ITERATION / P; x++)
{
  if((x%2)==0)
  {
    clSetKernelArg(k_stencil, 0, sizeof(cl_mem), &mobj_buf0);
    clSetKernelArg(k_stencil, 1, sizeof(cl_mem), &mobj_buf1);

    cl_event kernel_event;
    clEnqueueTask(queue, k_stencil, 0, NULL, &kernel_event);
    clWaitForEvents(1, &kernel_event);
    clReleaseEvent(kernel_event);
  }
  else
  {
    clSetKernelArg(k_stencil, 0, sizeof(cl_mem), &mobj_buf1);
    clSetKernelArg(k_stencil, 1, sizeof(cl_mem), &mobj_buf0);

    cl_event kernel_event;
    clEnqueueTask(queue, k_stencil, 0, NULL, &kernel_event);
    clWaitForEvents(1, &kernel_event);
    clReleaseEvent(kernel_event);
  }
}
```

Listing 6.3 The host code that executes the kernel

Board	p395_hpc_d8
Global Memory BW (MEMORY)	34112 MB/s

Source Code | Kernel Execution | stencil

File Name	Directory
stencil_395_prof.cl	/home bi_2D_5P/FPGA/VECT1/device/stencil_395_prof.cl

Line #	Source: stencil_395_prof.cl	Attributes	Stall%	Occupancy%	Bandwidth
21	}				
22	shiftreg[0][0] = (count < W*H) ? din[count] : 0.0f;	(__global·{MEMORY},read)	(0.0%)	(100.0%)	(984.9MB/s, 97.61%Efficiency)
59	dout[count-(j+1)*(W+2)] = result;	(__global·{MEMORY},write)	(0.0%)	(100.0%)	(961.4MB/s, 100.00%Efficiency)

Fig. 6.9 Profiling result for Listing 6.2

6.1.2.3 Performance Evaluation

We compiled the kernel for 100 parallel iterations (i.e., $P = 100$) using OpenCL SDK 16.0, and executed on Nallatech 395-D8 [8] FPGA board. The processing time is 16.8 s for 15,000 iterations and an 8192×8192 grid. Since the data transfers between the host and the device is done only at the beginning and at the end as shown in Fig. 6.6, the data transfer time is 0.08 s, and it is negligible compared to the total processing time.

Figure 6.9 shows the profiling results of the kernel. There is no-stall and the occupancy is 100%. Both load and store efficiencies are close to 100% so that all accessed data are used for computation (see Sect. 3.3.2). This is very close to the ideal profiling results we can expect from an optimized kernel. However, the memory-access bandwidth is less than 1 GBps which is only 10% of the theoretical bandwidth. This shows that the memory bandwidth is not a bottleneck and we may able to improve the performance by doing more stencil-parallel computations and accessing more data.

6.1.3 Using Loop Unrolling for Performance Improvement

As explained in Sect. 6.1.1, the processing time decreases as more iterations are processed in parallel. However, when the pipeline becomes deeper and deeper, the clock frequency tends to be lower. This is because of the clock skew problem for large circuits. As the clock frequency becomes lower, the processing time could be longer even if there are more parallel computations. On the other hand, if the FPGA board has a large global-memory bandwidth, we can increase stencil-parallel computations while decreasing the iteration-parallel computations. This way, we can maintain the same degree of parallelism, while keeping the clock frequency high.

To increase the degree of stencil-parallel computations, we can unroll the loop. Note that we cannot use the attribute `SIMD` in single work item kernels. Moreover, automatic unrolling using `#pragma unroll` is not possible since computations

are different between boundary and core cells. Therefore, we use manual unrolling. Stencil-parallel computation using loop-unrolling is shown in Listing 6.4. The loop is manually unrolled; the number of iteration-parallel computations is reduced to half by setting the number of loop-iterations to $H \cdot W/2 + P \cdot (W/2 + 2)$ as shown in line 7; two stencils are computed in one loop-iteration as shown from lines 36 to 49. The vector data type `float2` is used to ensure that the memory access is coalesced.

```
1  __attribute__((task))
2  __kernel void stencil2( __global const float2 * restrict din,
3                          __global float2 * restrict dout )
4  {
5    float2 zero = {0.0f, 0.0f};
6    float2 shiftreg[P][2*W/2+3];
7    const int loop_iterations = H*W/2 + P*(W/2+2);
8
9    int count = 0;
10   while (count != loop_iterations)
11   {
12     #pragma unroll
13     for (int i = 2*W/2+2; i>0; --i)
14     {
15       #pragma unroll
16       for (int j=0; j<P; j++ )
17       {
18         shiftreg[j][i] = shiftreg[j][i-1];
19       }
20     }
21     shiftreg[0][0] = (count < H*W/2) ? din[count] : zero;
22
23     float2 result;
24     float2 ce, le, ri, to, bo;
25     #pragma unroll
26     for (int j=0; j<P-1; j++)
27     {
28       to    = shiftreg[j][2*W/2+2];
29       le.s0 = shiftreg[j][1*W/2+3].s1;
30       le.s1 = shiftreg[j][1*W/2+2].s0;
31       ce    = shiftreg[j][1*W/2+2];
32       ri.s0 = shiftreg[j][1*W/2+2].s1;
33       ri.s1 = shiftreg[j][1*W/2+1].s0;
34       bo    = shiftreg[j][0*W/2+2];
35
36       if( ((count-(j+1)*(W/2+2)) % (W/2)) != 0 && count >= (j+1)*(W/2+2)+W/2
37           && count < ((j+1)*(W/2+2)+(H-1)*W/2) )
38         shiftreg[j+1][0].s0 = k1*to.s0 + k2*le.s0 + k3*ce.s0 + k4*ri.s0
39                               + k5*bo.s0;
40       else
41         shiftreg[j+1][0].s0 = 0.0f;
42
43       if( ((count-(j+1)*(W/2+2)) % (W/2)) != (W/2-1) &&
44           count >= (j+1)*(W/2+2)+W/2 && count < ((j+1)*(W/2+2)+(H-1)*W/2) )
45         shiftreg[j+1][0].s1 = k1*to.s1 + k2*le.s1 +  k3*ce.s1 + k4*ri.s1
46                               + k5*bo.s1;
47       else
48         shiftreg[j+1][0].s1 = 0.0f;
49     }
50
51     int j = P-1;
52     to    = shiftreg[j][2*W/2+2];
53     le.s0 = shiftreg[j][1*W/2+3].s1;
54     le.s1 = shiftreg[j][1*W/2+2].s0;
55     ce    = shiftreg[j][1*W/2+2];
56     ri.s0 = shiftreg[j][1*W/2+2].s1;
57     ri.s1 = shiftreg[j][1*W/2+1].s0;
58     bo    = shiftreg[j][0*W/2+2];
```

```
59
60      if(count >= (j+1)*(W/2+2))
61      {
62        if( ((count-(j+1)*(W/2+2)) % (W/2)) != 0 && count >= (j+1)*(W/2+2)+W/2
63            && count < ((j+1)*(W/2+2)+(H-1)*W/2) )
64          dout[count-(j+1)*(W/2+2)].s0 = k1*to.s0 + k2*le.s0 + k3*ce.s0
65                                       + k4*ri.s0 + k5*bo.s0;
66        else
67          dout[count-(j+1)*(W/2+2)].s0 = 0.0f;
68
69        if( ((count-(j+1)*(W/2+2)) % (W/2)) != (W/2-1) &&
70            count >= (j+1)*(W/2+2)+W/2 && count < ((j+1)*(W/2+2)+(H-1)*W/2) )
71          dout[count-(j+1)*(W/2+2)].s1 = k1*to.s1 + k2*le.s1 +  k3*ce.s1
72                                       + k4*ri.s1 + k5*bo.s1;
73        else
74          dout[count-(j+1)*(W/2+2)].s1 = 0.0f;
75      }
76      count++;
77    }
78  }
```

Listing 6.4 Increasing stencil-parallel computations using loop-unrolling

Figure 6.10 shows the profiling results. The stall is 0%, the occupancy percentage is 100, and the efficiency is close to 100%. These results are very similar to the profiling results for the previous kernel implementation in Sect. 6.1.2. However, the memory-access bandwidth is increased to nearly 2 GBps due to the increased data access caused by loop-unrolling. Since the throughput is still less than memory bandwidth, we can do more stencil-parallel computation.

Table 6.1 shows the comparison of the processing times of the naive implementation with no optimization and the one with loop-unrolling. The processing time is reduced by 8.4% for loop-unrolling. This is because of the clock frequency improvement of 8.4% due to the reduction of the iteration-parallel computations.

Table 6.2 shows the comparison of the resource utilization of the naive implementation with no optimization and the implementation with loop-unrolling. There is no significant difference in logic, DSP, or register utilization. However, we can see significant differences in the memory and RAM block utilizations. In both implementations, the size of the shift-register in a PCM is the same. However, when loop-unrolling is used, the number of iteration-parallel computations is reduced to half. Therefore, the total number of shift-registers in all PCMs is reduced to half.

Line #	Source: stencil2_395_prof.cl	Attributes	Stall%	Occupancy%	Bandwidth
22	}				
23	shiftreg[0][0] = (count < H*W/2) ? din[count] : zero;	(__global{MEMORY},read)	(0.0%)	(100.0%)	(1959.2MB/s, 98.79%Efficiency)
73	dout[count-(j+1)*(W/2+2)].s1 = k1*to.s1 + k2*le.s...	(__global{MEMORY},write)	(0.0%)	(100.0%)	(1935.5MB/s, 100.00%Efficiency)

Fig. 6.10 Memory access is very efficient with no stall

Table 6.1 Comparison of the processing times of the naive implementation with no optimization and the implementation with loop-unrolling

Implementation	Processing time (s)	Clock frequency (MHz)
Naive (no optimization)	11.42	226.6
Using loop-unrolling	10.45	245.7

Table 6.2 Comparison of the resource utilization of the naive implementation with no optimization and the implementation with loop-unrolling

Implementation	Logic	DSP	Memory (MB)	RAM blocks	Registers
Naive (no optimization)	236,446	500	3.47	2234	423,449
Using loop-unrolling	228,340	500	1.90	1513	415,451

Since a shift-register is constructed using registers and RAM blocks, utilizations of these resources are reduced. As a result, the clock frequency is improved.

6.1.4 *Dividing a Large Kernel into Multiple Small Kernels for Performance Improvement*

As mentioned in Sect. 6.1.3, the clock frequency tends to be low for large circuits. To solve this problem, we can divide a large kernel into multiple small ones. Listing 6.5 shows the kernel code of this implementation. It contains two kernels, `stencil_k1` and `stencil_k2`. We use the channel `ch_data` to connect the kernels. The first kernel (`stencil_k1`) writes its output to the channel instead of the global memory. The second kernel (`stencil_k2`) reads its input from the same channel.

```
channel float ch_data __attribute__((depth(8)));

__attribute__((task))
__kernel void stencil_k1( __global const float * restrict din )
{
  float shiftreg[P1][2*W+4];
  const int loop_iteration = P1*(W+2) + W*H;
  int count = 0;

  while (count != loop_iteration)
  {
    #pragma unroll
    for (int i = 2*W+3; i>0; --i)
    {
      #pragma unroll
      for (int j=0; j<P1; j++ )
      {
        shiftreg[j][i] = shiftreg[j][i-1];
      }
    }
    shiftreg[0][0] = (count < W*H) ? din[count] : 0.0f;

    float result;
    float ce, le, ri, to, bo;
    #pragma unroll
    for (int j=0; j<P1-1; j++)
    {
      to = shiftreg[j][2*W+2];
      le = shiftreg[j][1*W+3];
      ce = shiftreg[j][1*W+2];
      ri = shiftreg[j][1*W+1];
      bo = shiftreg[j][0*W+2];

```

```
      if( count >= ((j+1)*(W+2)+W) && (((count-(j+1)*(W+2)) % W) != 0) &&
        ((count-(j+1)*(W+2)) % W) != (W-1) && count < ((j+1)*(W+2)+W*(H-1)) )
        result = k1*to + k2*le + k3*ce + k4*ri + k5*bo;
      else
        result = 0.0f;

      shiftreg[j+1][0] = result;
    }

    int j = P1-1;
    to = shiftreg[j][2*W+2];
    le = shiftreg[j][1*W+3];
    ce = shiftreg[j][1*W+2];
    ri = shiftreg[j][1*W+1];
    bo = shiftreg[j][0*W+2];

    if(count >= (j+1)*(W+2))
    {
      if( count >= ((j+1)*(W+2)+W) && (((count-(j+1)*(W+2)) % W) != 0) &&
        ((count-(j+1)*(W+2)) % W) != (W-1) && count < ((j+1)*(W+2)+W*(H-1)) )
        result = k1*to + k2*le + k3*ce + k4*ri + k5*bo;
      else
        result = 0.0f;

      write_channel_altera(ch_data, result);
    }
    count++;
  }
}

__attribute__((task))
__kernel void stencil_k2( __global float * restrict dout )
{
  float shiftreg[P2][2*W+4];
  const int loop_iteration = P2*(W+2) + W*H;
  int count = 0;

  while (count != loop_iteration)
  {
    #pragma unroll
    for (int i = 2*W+3; i>0; --i)
    {
      #pragma unroll
      for (int j=0; j<P2; j++ )
      {
        shiftreg[j][i] = shiftreg[j][i-1];
      }
    }
    shiftreg[0][0] = (count < W*H) ? read_channel_altera(ch_data) : 0.0f;

    float result;
    float ce, le, ri, to, bo;
    #pragma unroll
    for (int j=0; j<P2-1; j++)
    {
      to = shiftreg[j][2*W+2];
      le = shiftreg[j][1*W+3];
      ce = shiftreg[j][1*W+2];
      ri = shiftreg[j][1*W+1];
      bo = shiftreg[j][0*W+2];

      if( count >= ((j+1)*(W+2)+W) && (((count-(j+1)*(W+2)) % W) != 0) &&
        ((count-(j+1)*(W+2)) % W) != (W-1) && count < ((j+1)*(W+2)+W*(H-1)) )
        result = k1*to + k2*le + k3*ce + k4*ri + k5*bo;
      else
        result = 0.0f;

```

```
      shiftreg[j+1][0] = result;
    }

    int j = P2-1;
    to = shiftreg[j][2*W+2];
    le = shiftreg[j][1*W+3];
    ce = shiftreg[j][1*W+2];
    ri = shiftreg[j][1*W+1];
    bo = shiftreg[j][0*W+2];

    if(count >= (j+1)*(W+2))
    {
      if( count >= ((j+1)*(W+2)+W) && (((count-(j+1)*(W+2)) % W) != 0) &&
        ((count-(j+1)*(W+2)) % W) != (W-1) && count < ((j+1)*(W+2)+W*(H-1)) )
        result = k1*to + k2*le + k3*ce + k4*ri + k5*bo;
      else
        result = 0.0f;

      dout[count-(j+1)*(W+2)] = result;
    }
    count++;
  }
}
```

Listing 6.5 Dividing a large kernel into multiple small ones

In this implementation, we execute both kernels concurrently. The execution commands for concurrently executed kernels must be put into different command queues. For this purpose, we put the execution commands for the two kernels into two different command queues as follows.

```
clEnqueueTask(queue[0], k_stencil[0], 0, NULL, &kernel_event[0]);
clEnqueueTask(queue[1], k_stencil[1], 0, NULL, &kernel_event[1]);
```

Table 6.3 shows the comparison of the processing times of the naive implementation and the two-kernel implementation. The processing time is reduced by 7.6% in the two-kernel implementation. This is because the offline compiler can optimize the smaller kernel for a higher clock frequency. Table 6.4 shows the comparison of the resource utilizations of the naive implementation and the two-kernel implementation. There is no significant difference in logic, DSP, memory, RAM block, and register utilization.

6.1.5 Optimization Using Compiler Options

The memory interleaving causes the data to be loaded from all memory banks interchangeably to balance the memory-access bandwidths across all memory banks, as explained in Sect. 5.3.2. This requires a complex datapath. To simplify

Table 6.3 Comparison of the processing times of the naive implementation and the two-kernel implementation

Implementation	Processing time (s)	Clock frequency (MHz)
Naive (no optimization)	11.42	226.6
Dividing into two kernels	10.55	245.2

Table 6.4 Comparison of the resource utilization of the naive implementation with no optimization and the two-kernel implementation

Implementation	Logic	DSP	Memory (MB)	RAM blocks	Registers
Naive (no optimization)	236,446	500	3.47	2234	423,449
Dividing into two kernels	236,395	500	3.46	2237	426,136

Table 6.5 Comparison of the processing times of the naive implementation and the implementation with non-interleaved memory access

Implementation	Processing time (s)	Clock frequency (MHz)
Naive (no optimization)	11.42	226.6
Using non-interleaved memory access	11.02	234.7

Table 6.6 Resource utilization of the naive implementation and the implementation with non-interleaved memory access

Implementation	Logic	DSP	Memory (MB)	RAM blocks	Registers
Naive (no optimization)	236,446	500	3.47	2234	423,449
Using non-interleaved memory access	235,547	500	3.45	2182	421,492

the memory access and to reduce the resource overhead, we can use non-interleaved memory access. The stencil-computation kernel has one read-only input and one write-only output. Therefore, we can use non-interleaved access where one memory bank is used as read-only and the other is used as write-only. Since the size of the input and output data is the same, throughput is balanced between two memory banks. The compiler option `--no-interleaving default` is used to compile the kernel with non-interleaved memory access.

Table 6.5 shows the comparison of the processing times of the naive implementation and the implementation with non-interleaved memory access. The processing time is reduced slightly by 3.5% in the implementation with non-interleaved memory access. This is because of the small improvement in the clock frequency due to simplified data paths. Table 6.6 shows the comparison of the resource utilizations of the naive implementation and the implementation with non-interleaved memory access. There is no significant difference in resource utilization.

We can also use compiler options `--fp-relaxed` and `--fpc` to allow the compiler to re-arrange the computation order, and to use fused floating-point operations respectively, as explained in Sect. 4.2.2. This may reduce the resource utilization and improve the performance although this changes the computation results slightly.

Table 6.7 Comparison of the processing times of different implementations

Implementation	Processing time (s)	Clock frequency (MHz)
Naive (no optimization)	11.42	226.6
Using loop-unrolling	10.45	245.7
Dividing into two kernels	10.55	245.2
Using non-interleaved memory access	11.02	234.7
Using loop-unrolling with non-interleaved memory access	10.21	251.4

Table 6.8 Resource utilization of different implementations

Implementation	Logic	DSP	Memory (MB)	RAM blocks	Registers
Naive (no optimization)	236,446	500	3.47	2234	423,449
Using loop-unrolling	228,340	500	1.90	1513	415,451
Dividing into two kernels	236,395	500	3.46	2237	426,136
Using non-interleaved memory access	235,547	500	3.45	2182	421,492
Using loop-unrolling with non-interleaved memory access	228,393	500	1.90	1513	415,391

6.1.6 Summary of the Results of Different Strategies for Performance Improvement

Table 6.7 shows the comparison of the processing times of different implementations. Table 6.8 shows the comparison of the resource utilizations of different implementations. Every implementation has the same degree of parallelism without any memory access bottlenecks. Therefore, the number of clock cycles required by all method is the same. However, the clock frequency is different among the implementations. The implementation that uses loop-unrolling with non-interleaved memory access has the highest clock frequency and also the shortest processing time. This is because of its smallest circuit.

6.2 Design of a Heterogeneous Computing System for Molecular Dynamic Simulations

Molecular dynamics (MD) simulations are very important to study physical properties of atoms and molecules [9, 10]. However, a huge processing time is required even to simulate a few nano-seconds of an actual experiment. Therefore, hardware acceleration is very important.

There are many widely used software packages for MD simulations such as AMBER [11], Desmond [12], GROMACS [13], LAMMPS [14], CHARMM [15], etc. Hardware acceleration is also used to reduce the huge processing time in

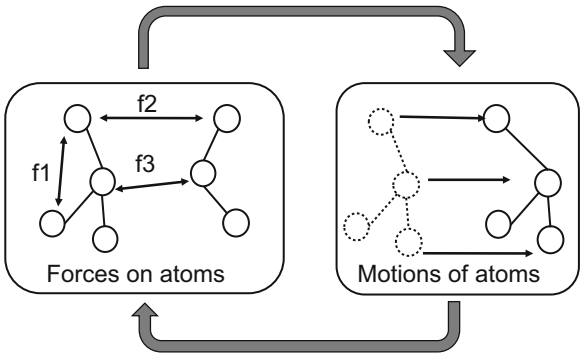


Fig. 6.11 Molecular dynamics simulation model

MD simulations. ASIC (application specific integrated circuit) implementations of MD simulation are proposed in [16–18]. However, designing such special purpose processors takes a long design time and also involves a huge financial cost. Another way of hardware acceleration is provided by FPGAs [19–21]. In this chapter, we explain how an FPGA-based heterogeneous computing system is designed in [7, 22] to accelerate MD simulations.

6.2.1 Molecular Dynamics Simulation

Atoms and molecules move due to various forces acting on those [23, 24]. By calculating the forces, we can find the movements. Therefore, MD simulations contain two major steps, the force computation and the motion update as shown in Fig. 6.11. When atoms move, their positions get updated. As a result, we have to compute the forces again using new atom positions. Due to the movements of atoms, the force computation and the motion update are done again and again for many iterations. Therefore, MD simulation is considered as an iterative computation. It requires millions of iterations to simulate a few nanoseconds of the real time. Note that the motion update is done using the classical Newtonian physics. When we know the force, the acceleration, and the initial velocity, we can find the distance that molecules move in a small time period using Newton's equations.

The forces are caused by the interactions among atoms. Figure 6.12 shows various interactions. These interactions are classified into bonded and non-bonded interactions. The bonded interactions are the acts between atoms that are linked by covalent bonds. Bonded interactions cause stretching, angle, torsion, as shown in Fig. 6.12a. Forces due to bonded interactions are called bonded forces. Bonded forces affect only a few neighboring atoms, and can be computed in $O(N)$ time for N atoms. Non-bonded interactions are the acts between atoms which are not linked by covalent bonds. As shown in Fig. 6.12b, Lennard Jones potential and electrostatic potential are such interactions and cause non-bonded forces. Since those forces exist among all atoms, the computation requires $O(N^2)$ processing time. Therefore,

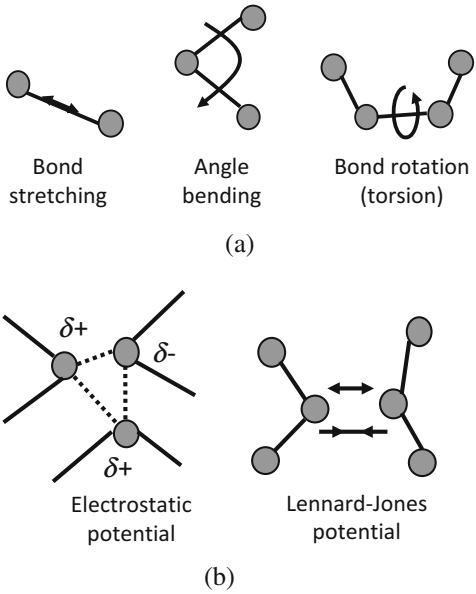


Fig. 6.12 Forces considered in MD simulations. (**a**) Bonded interactions. (**b**) Non-bonded interactions

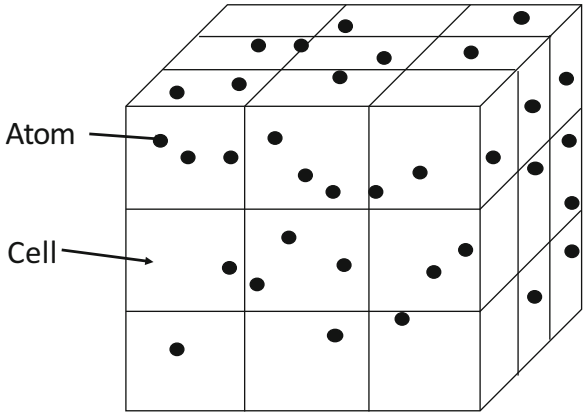


Fig. 6.13 Cells and atoms

the non-bonded force computation requires a large processing time, and usually accounts for more than 90% of the total processing time required for the simulation.

MD simulations use a system that is usually represented by a box, which is divided into multiple cells as shown in Fig. 6.13. Atoms and molecules move through the cells. To reduce the computation amount, we can ignore the non-bonded forces between distant cells that are very far from each other. This distance is called the "cut-off" distance. Let us explain how to compute the forces between two atoms. First we extract a list of cell-pairs within the cut-off distance. We call this a "cell-pair list." For each cell-pair, we compute the non-bonded forces between two atoms. Note that the same cell also paired to itself.

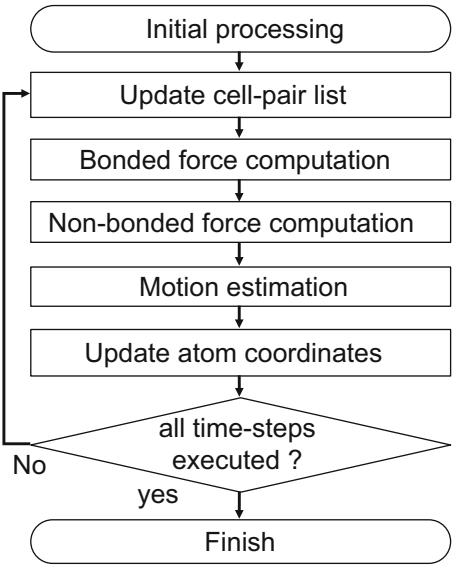


Fig. 6.14 Flowchart of the MD simulation [25]

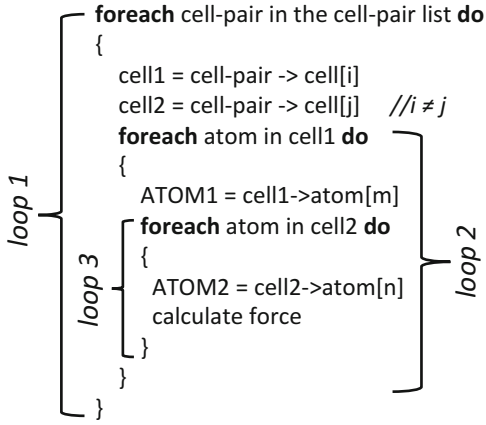


Fig. 6.15 Non-bonded force computation code

6.2.2 *OpenCL-Based System*

Figure 6.14 shows the flowchart of the MD simulation [25]. The non-bonded force computation is the most time-consuming part of the MD simulation due to its large computation amount. Therefore, we allocate this task to the device (FPGA board). The other tasks do not take much time and we allocate those to the host (CPU).

Figure 6.15 shows a simplified version of the kernel code for non-bonded force computations [26]. It contains three loops. The iterations of *loop 1* are executed for each element in the cell-pair list. The cells in a cell-pair are *cell1* and *cell2*. The *loop 2* and the *loop 3* compute the forces between all atom-pairs of *cell1* and *cell2*.

```
 + Loop "Block4" (file calc_in_step.cl line 31)
 | NOT pipelined due to:
 |
 |   Loop iteration ordering: iterations may get out of order with respect to the listed
inner loops,
 |   as the number of iterations of the listed inner loops may be different for different
iterations of this loop.
 |       Loop "Block6" (file calc_in_step.cl line 57)
 |       Loop "Block7" (file calc_in_step.cl line 68, line 68, line 70)
 |
 |   To fix this, make sure the listed inner loops have the same number of iterations for
each iteration of this loop.
 |   See "Out-of-Order Loop Iterations" section of the Best Practices Guide for more
information.
 |   Not pipelining this loop will most likely lead to poor performance.
 |
 |
 |-+ Loop "Block5" (file calc_in_step.cl line 36)
   | NOT pipelined due to:
   |
   |   Loop iteration ordering: iterations may get out of order with respect to the listed
inner loops,
   |   as the number of iterations of the listed inner loops may be different for different
iterations of this loop.
   |       Loop "Block6" (file calc_in_step.cl line 57)
   |       Loop "Block7" (file calc_in_step.cl line 68, line 68, line 70)
   |
   |   To fix this, make sure the listed inner loops have the same number of iterations for
each iteration of this loop.
   |   See "Out-of-Order Loop Iterations" section of the Best Practices Guide for more
information.
   |   Not pipelining this loop will most likely lead to poor performance.
   |
   |
   |-+ Loop "Block6" (file calc_in_step.cl line 57)
     | NOT pipelined due to:
     |
     |   Loop iteration ordering: iterations may get out of order with respect to the
listed inner loop,
     |   as the number of iterations of the listed inner loop may be different for
different iterations of this loop.
     |       Loop "Block7" (file calc_in_step.cl line 68, line 68, line 70)
     |
     |   To fix this, make sure the listed inner loop has the same number of iterations for
each iteration of this loop.
     |   See "Out-of-Order Loop Iterations" section of the Best Practices Guide for more
information.
     |   Not pipelining this loop will most likely lead to poor performance.
```

Fig. 6.16 Compilation report for the code in Fig. 6.15

The numbers of loop-iterations of the *loop 2* and the *loop 3* equal to the number of atoms of *cell1* and *cell2*, respectively. Since atoms move, the number of atoms per cell varies from cell-to-cell and also from iteration-to-iteration. Therefore, all the three loops do not have fixed (or static) loop boundaries.

Figure 6.16 shows the compilation report for the code shown in Fig. 6.15. A pipelined architecture is not implemented since the loop-boundaries vary for different cells and different iterations. Such a non-pipelined architecture is not an efficient one.

Table 6.9 shows the processing time of the FPGA implementation. The details of the evaluation results are reported in [27]. The processing time is measured for a molecular dynamics simulation that contains 22,795 atoms. The width, height, and the depth of the simulation box are 61.24×10^{-10} m. The CPU is Intel Xeon

Table 6.9 Processing time of non-bonded force computation for the code shown in Fig. 6.15

Implementation	Processing time (s)
CPU (Xeon E5-1650 v3)	0.68
FPGA (Aria 10 10AX115N3F45I2SG): using the same CPU code	88.03

E5-1650 v3. The FPGA board is Terasic DE5a-Net Arria 10 FPGA Development Kit [28] that contains an Aria 10 10AX115N3F45I2SG FPGA. We used Quartus Prime Pro 16.1 compiler with SDK for OpenCL. According to the results, the processing time of the accelerator is more than 100 times larger compared to that of a CPU. The reason for the large processing time is the non-pipelined implementation of the loops.

6.2.3 Improving Performance by Removing Nested-Loops

To reduce the processing time, we have to design a pipelined architecture. For this purpose, the loop boundaries should be well defined at the compilation stage. The works in [22, 27] propose a method to solve this problem. In this method, the force computation is separated from the atom-pair selection. The complete atom-pairs list is extracted based on the cell-pair list. Then the force computation is performed for each atom-pair in the list. The atom-pair-list extraction is just a searching procedure that does not contain heavy computations. On the other hand, force computation contains many multiplications and divisions. Therefore, the host is used for atom-pair-list extraction. This list is transferred to the device for force computation. Once the list is available, only a single loop is sufficient for the force computation of all the atom-pairs in the list. As a result, the offline compiler can implement a pipelined datapath to accelerate the computation.

Figure 6.17 shows the task allocation for heterogeneous computing proposed in [22, 27]. To perform non-bonded force computation in the device, we have to transfer the atom-pair list and atom coordinates to the device. We also have to transfer the computed results of force from the device to the host for motion estimation. This data transfers impose additional time on the total processing time.

Table 6.10 shows the processing time of the non-bonded force computation using the method proposed in [22, 27]. The processing time of one iteration is shown. Since the offline compiler generates a pipelined datapath, parallel processing is done. As a result, the processing time on the device is reduced to 0.17 s. This gives a speed-up of four times compared to the processing on the host.

Table 6.11 shows the resource usage of the FPGA accelerator. As shown in the table, only 16% of the FPGA resources are used. To obtain further speed-ups, we can use more FPGA resources for parallel computation. Note that, a wide bandwidth is also required as well as the computation resources. If we use 80% of the FPGA resources and 50 GBps of bandwidth, we can theoretically obtain a speed-up of

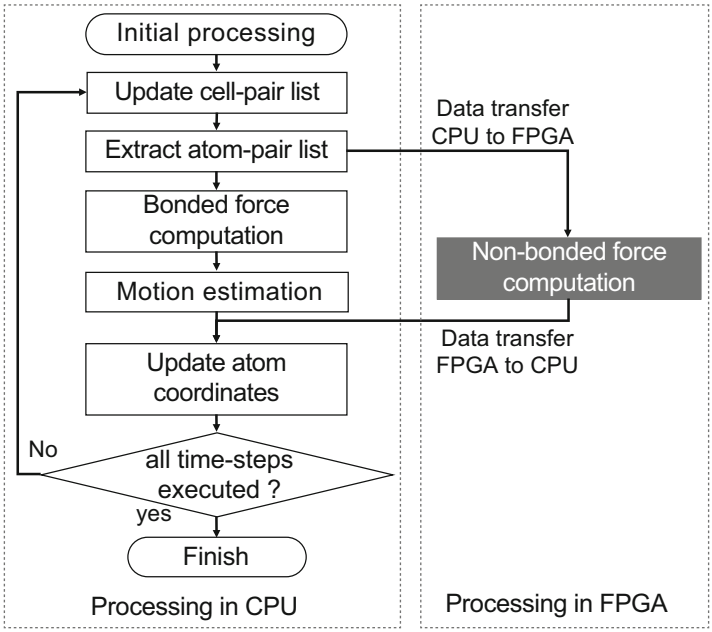


Fig. 6.17 Heterogeneous computation proposed in [22, 27] for MD simulations

Table 6.10 Processing time of non-bonded force computation using the method proposed in [22, 27]

Implementation	Processing time (s/iteration)
CPU (Xeon E5-1650 v3)	0.68
FPGA (Aria 10 10AX115N3F45I2SG)	0.17

Table 6.11 Resource usage of the FPGA accelerator

Resource	Usage	Percentage used (%)
Logic (ALMs)	68,163	16.0
Registers	120,168	7.0
Memory (Mbits)	3.35	3.5
DSPs	57	3.8

24.5 times. Fortunately, there are many recent FPGA boards that have such a large bandwidth [29].

Table 6.12 shows the comparison of the total processing time including data transfers between the host and the device. The total processing time of the FPGA-based heterogeneous computing system is 0.50 s. The data transfers between the CPU and the FPGA account for 66% of the total processing time. Due to these data transfers, the speed-up compared to CPU processing is only 1.6 times.

Table 6.12 Comparison of the total processing times

Implementation	Processing time (s/iteration)
Total processing time (heterogeneous processing on CPU and FPGA)	0.50
Non-bonded force computation in FPGA[a]	(0.17)
Bonded force computation in CPU[a]	(0.14)
Data transfer: CPU to FPGA[a]	(0.25)
Data transfer: FPGA to CPU[a]	(0.08)
Total processing time in conventional method (using CPU only)	0.82

[a]Portions of the total processing time of the FPGA

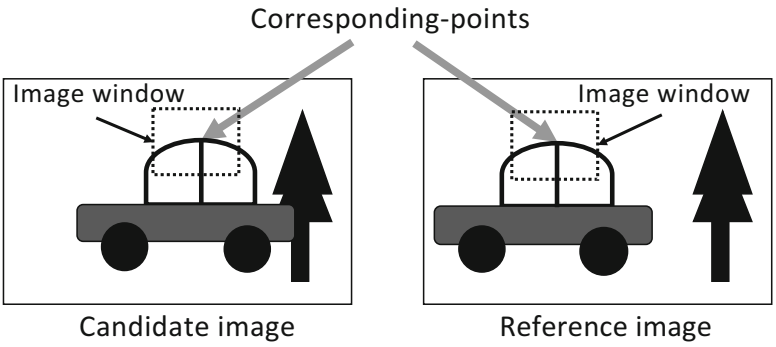


Fig. 6.18 Image matching

Such data transfer problems can be solved by using SoC-type FPGAs which has a CPU cores and an FPGA core in the same chip. As a result, we can use on-chip data transfers, instead of sending data through the PCIe. On-chip data transfers are much faster compared to the PCIe based transfers. Moreover, next generation SoC-type FPGAs support unified memory where the same memory is shared by CPU cores and the FPGA cores. In this case, we do not require data transfers at all. That will significantly reduce the total processing time.

6.3 Design of a Highly-Accurate Image Matching Accelerator

6.3.1 Image Matching Using Phase Only Correlation

Image matching [30] is a process of finding corresponding points between two images as shown in Fig. 6.18. It is an important and fundamental task in various image processing applications such as stereo vision [31], motion analysis [32], etc. Phase-only correlation (POC) [33] is a correspondence search method with a high

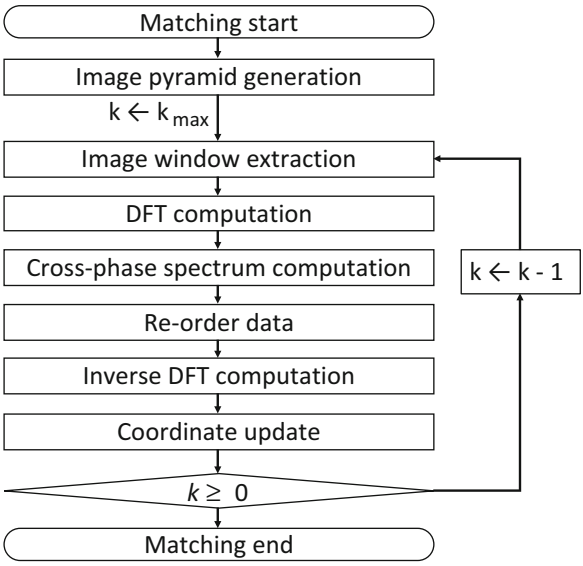


Fig. 6.19 Flowchart of the POC based image matching algorithm

degree of accuracy. The correspondence search is done in the frequency domain so that very accurate image matching at sub-pixel level is possible. Discrete Fourier transforms (DFTs) is used to get the frequency information of the images. Note that, POC uses only the phase component [34]. POC is used in various applications such as 3-D measurement [35] and biometrics [36].

The correspondence search is done by employing a coarse-to-fine searching strategy [37]. To realize this strategy, we generate a multi-layer image pyramid. The resolutions of the images in the layers increase from coarse-to-fine. The image matching starts from the layer that contains the coarsest images and gradually moves to the layers with finer images. We determine the corresponding points roughly using coarser images and increase the accuracy gradually as we proceed to the finer images. This approach can limit the search area of the finer images and can reduce the computation amount by using the matching results of the coarser images.

Figure 6.19 shows the flowchart of the image matching algorithm that contains multiple procedures. We start from the coarsest images of layer $k_{\max}$. We apply DFT to the image window to get the frequency data. Matching is performed in the frequency domain and inverse DFT is applied to get the matching coordinates. The coordinates are used in the matching of the next layer to limit the search area. Search is completed after processing the images of all layers.

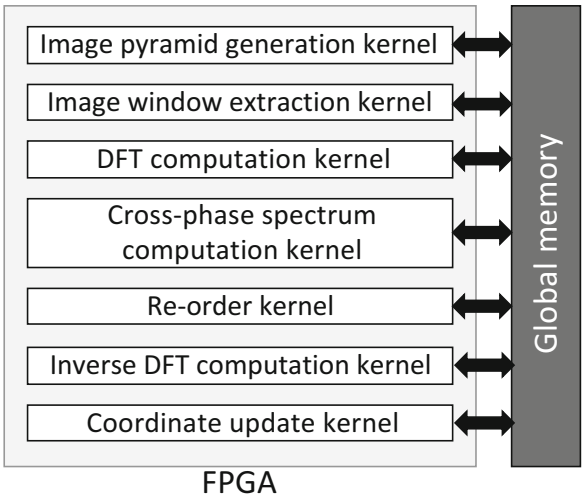


Fig. 6.20 FPGA implementation of the POC method

6.3.2 OpenCL-Based Implementation

As shown in Fig. 6.19, the output of one procedure is used as the input of the next procedure. Therefore, we have to transfer data from one procedure to the next. Figure 6.20 shows the simplest implementation of this algorithm. Each procedure is implemented as a separate kernel. This is because the use of multiple small kernels is important to increase the clock frequency as explained in Sect. 6.1.4. The inputs and outputs of the kernels are loaded from and stored to the global memory, respectively. The global memory is used to transfer the data between kernels. This method is very similar to the typical GPU-based processing where the data of different kernels are transferred using the global memory.

Table 6.13 shows the comparison of the processing times of the FPGA and the CPU. We used Nallatech 395-D8 FPGA board [8] that contains a Stratix V 5SGSED8N2F46C2LN FPGA. The CPU is Intel Core i7-3960X. The CPU code is fully parallelized to use all six cores. The processing time of the FPGA accelerator is almost 1.8 times longer than that of the CPU. The reason for the large processing time on the FPGA is due to the memory-access-bottleneck. Since all kernels compete for the global memory access, the required bandwidth exceeds the theoretical bandwidth. The bandwidth of FPGAs is smaller compared to those of CPUs and GPUs. In fact, the FPGA board we used has the theoretical bandwidth of 34.1 GBps which is much smaller compared to 51.2 GBps memory bandwidth of the CPU.

Table 6.13 Processing time comparison of the CPU and FPGA implementations

Implementation	Processing time (s)
CPU (Intel Core i7-3960X @ 3.3GHz)	49.09
FPGA (Stratix V 5SGSED8N2F46C2LN)	88.19

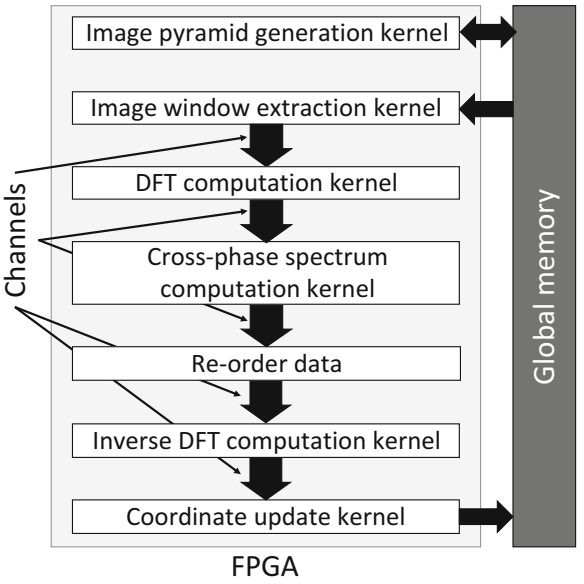


Fig. 6.21 FPGA implementation of the POC method using channels

6.3.3 Reducing the Required Bandwidth Using Channels

To improve the performance, we have to reduce the global memory access. For this purpose, we can use channels to transfer data between kernels as explained in Sect. 5.7. Recall that the channels are FIFO buffers that are configured using on-chip RAM blocks and registers.

Figure 6.21 shows the implementation using channels. The channels are used to transfer the output of one kernel to the next kernel as its inputs without accessing the global memory. As a result, the global memory access is reduced considerably compared to the one in Fig. 6.20.

For further reduction of the global memory access, we can use the channels more aggressively. In the POC algorithm, the coordinates of the corresponding-points of the previous layer must be preserved until the start of the computation of the next layer. Since there are 100×100 corresponding-points, and one corresponding-point is computed in one clock cycle, the lifetime is 10,000 cycles. In order to store the data for such a long time, we use a buffered channel as explained in Sect. 5.7. The buffer size is specified by the `depth` attribute. We set `depth` to the same value as the lifetime, 10,000. Figure 6.22 shows this implementation. Since the coordinates of the corresponding-point are transferred through the channels, the required global-memory bandwidth is reduced.

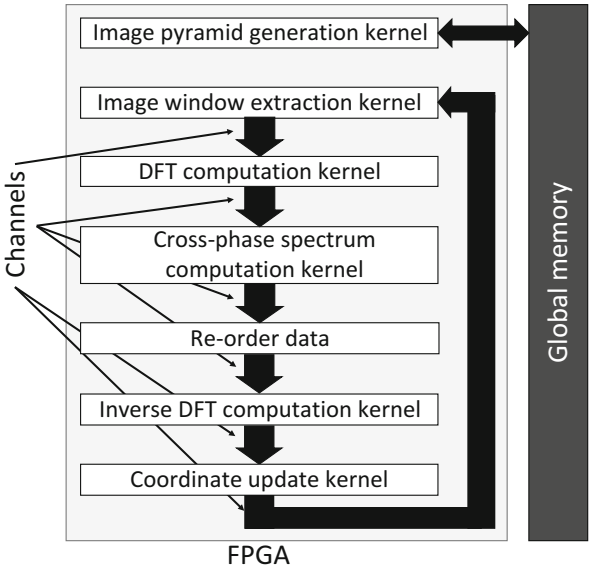


Fig. 6.22 FPGA implementation of the POC method using channels more aggressively

Table 6.14 Power consumption of the CPU and GPU and FPGA implementations

Implementation	Processing time (s)
CPU (Intel Core i7-3960X @ 3.3GHz)	49.09
GPU (Geforce GTX 680)	11.51
FPGA (without channels)	88.19
FPGA (with channels)	10.39

Table 6.15 Processing time comparison of the CPU and FPGA implementations

System	Power consumption (W)
CPU (i7-3960X)	147
GPU (GTX 680) and CPU (i7-3960X)	130
FPGA (Stratix V) and CPU (i7-3960X)	16

Table 6.14 shows the processing time comparison of the CPU, GPU, and FPGA implementations. When we use channels, the processing time is reduced to 11% of that of the FPGA implementation with only global memory. The processing time of the FPGA implementation with channels is reduced to 21% of that of the CPU implementation. The processing time of FPGA is reduced to 90% of that of the GPU implementation.

Table 6.15 shows the power consumption of each system. The power consumption is defined as the difference of the power consumptions in the execution state and the idle state. The power consumption of the FPGA-based system is only 12.3% of that of the GPU-based system. Therefore, we can use OpenCL to design systems for real-world-applications with low-power and high-performance.

References

1. G. Karniadakis, S. Sherwin, *Spectral/hp Element Methods for Computational Fluid Dynamics* (Oxford University Press, Oxford, 2013)
2. K.S. Yee, Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. IEEE Trans. Antennas Propag. **14**(3), 302–307 (1966)
3. W.M. Kahan, Gauss-Seidel methods of solving large systems of linear equations, Ph.D. Thesis, University of Toronto, 1958
4. W. Luzhou, K. Sano, S. Yamamoto, Domain-specific language and compiler for stencil computation on FPGA-based systolic computational-memory array, in *Reconfigurable Computing: Architectures, Tools and Applications* (Springer, Berlin 2012), pp. 26–39
5. K. Sano, Y. Hatsuda, S. Yamamoto, Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth. IEEE Trans. Parallel Distrib. Syst. **25**(3) 695–705 (2014)
6. K. Dohi, K. Okina, R. Soejima, Y. Shibata, K. Oguri, Performance modeling of stencil computing on a stream-based FPGA accelerator for efficient design space exploration. IEICE Trans. Inf. Syst. **E98-D**(2), 298–308 (2015)
7. H.M. Waidyasooriya, M. Hariyama, K. Kasahara, Architecture of an FPGA accelerator for molecular dynamics simulation using OpenCL, in *Proceedings of the 15th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2016)* (2016), pp. 115–119
8. Nallatech 395 – with Stratix V D8 (2017), http://www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/395-d8/
9. C.Z. Wang, K.M. Ho, Material simulations with tight-binding molecular dynamics. J. Phase Equilib. **18**(6), 516–529 (1997)
10. V. Daggett, Protein folding-simulation. Chem. Rev. **106**(5), 1898–1916 (2006)
11. D.A. Case, T.E. Cheatham, T. Darden, H. Gohlke, R. Luo, K.M. Merz, A. Onufriev, C. Simmerling, B. Wang, R.J. Woods, The amber biomolecular simulation programs. J. Comput. Chem. **26**(16), 1668–1688 (2005)
12. K.J. Bowers, E. Chow, H. Xu, R.O. Dror, M.P. Eastwood, B.A. Gregersen, J.L. Klepeis, I. Kolossvary, M.A. Moraes, F.D. Sacerdoti, et al., Scalable algorithms for molecular dynamics simulations on commodity clusters, in *Proceedings of ACM/IEEE SC Conference* (2006), p. 43
13. S. Pronk, S.Pall, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M.R. Shirts, J. C. Smith, P. M. Kasson, D. van der Spoel, et al., Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. Bioinformatics **29**(7), 845–854 (2013)
14. S. Plimpton, P. Crozier, A. Thompson, *LAMMPS-Large-Scale Atomic/Molecular Massively Parallel Simulator*, vol. 18 (Sandia National Laboratories, Albuquerque, 2007)
15. B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D.J. States, S. Swaminathan, M. Karplus, CHARMM: a program for macromolecular energy, minimization, and dynamics calculations. J. Comput. Chem. **4**(2), 187–217 (1983)
16. T. Narumi, Y. Ohno, N. Okimoto, A. Suenaga, R. Yanai, M. Taiji, A high-speed special-purpose computer for molecular dynamics simulations: MDGRAPE-3, in *NIC Workshop*, vol. 34 (2006), pp. 29–36
17. D.E. Shaw, M.M. Deneroff, R.O. Dror, J.S. Kuskin, R.H. Larson, J.K. Salmon, C. Young, B. Batson, K.J. Bowers, J.C. Chao, et al., Anton, a special-purpose machine for molecular dynamics simulation. Commun. ACM **51**(7), 91–97 (2008)
18. D.E. Shaw, J. Grossman, J.A. Bank, B. Batson, J.A. Butts, J.C. Chao, M.M. Deneroff, R.O. Dror, A. Even, C.H. Fenton, et al., Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), pp. 41–53
19. E. Cho, A.G. Bourgeois, F. Tan, An FPGA design to achieve fast and accurate results for molecular dynamics simulations. *Parallel and Distributed Processing and Applications* (Springer, Berlin, 2007), pp. 256–267

20. M. Chiu, M.C. Herbordt, Molecular dynamics simulation high-performance reconfigurable computing systems. ACM Trans. Reconfigurable Technol. Syst. **3**(4), 23:1–23:37 (2010)
21. M. A. Khan, Scalable molecular dynamics simulation using FPGAs and multicore processors, Ph.D. Thesis, Boston University College of Engineering, 2013
22. H.M. Waidyasooriya, M. Hariyama, K. Kasahara, An FPGA accelerator for molecular dynamics simulation using OpenCL. Int. J. Networked Distrib. Comput. **5**(1), 52–61 (2017)
23. D.C. Rapaport, *The Art of Molecular Dynamics Simulation* (Cambridge University Press, Cambridge, 2004)
24. F. Jensen, *Introduction to Computational Chemistry* (Wiley, New York, 2013)
25. mypresto (2015), http://presto.protein.osaka-u.ac.jp/myPresto4/index.php?lang=en
26. T. Mashimo, Y. Fukunishi, N. Kamiya, Y. Takano, I. Fukuda, H. Nakamura, Molecular dynamics simulations accelerated by GPU for biological macromolecules with a non-Ewald scheme for electrostatic interactions. J. Chem. Theory Comput. **9**(12), 5599–5609 (2013)
27. H.M. Waidyasooriya, M. Hariyama, K. Kasahara, OpenCL-based implementation of an FPGA accelerator for molecular dynamics simulation. Inf. Eng. Express **3**(2), 11–23 (2017)
28. Terasic, DE5a-Net Arria 10 FPGA Development Kit (2017). https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=228&No=970&PartNo=2
29. Nallatech 395 – with Stratix V D8 (2017), http://www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/nallatech-510t-fpga-computing-acceleration-card/
30. R. Szeliski, *Computer Vision: Algorithms and Applications* (Springer, New York, 2011)
31. B.D. Lucas, T Kanade, An iterative image registration technique with an application to stereo vision, in *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81)* (1981), pp. 674–679
32. B.K.P. Horn, G.S. Brian, Determining optical flow. Artif. Intell. 17(1–3), 185–203 (1981)
33. K. Takita, T. Aoki, Y. Sasaki, T. Higuchi, K. Kobayashi, High-accuracy subpixel image registration based on phase-only correlation. IEICE Trans. Fundam. **E86-A**(8), 1925–1934 (2003)
34. C.D. Kuglin, D.C. Hines, The phase correlation image alignment method, in *Proceedings of the International Conference on Cybernetics and Society* (1975), pp.163–165
35. M.A. Muquit, S. Takuma, T. Aoki, A high-accuracy passive 3D measurement system using phase-based image matching. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **89**(3), 686–697 (2006)
36. K. Miyazawa, K. Ito, T. Aoki, K. Kobayashi, H. Nakajima, An effective approach for iris recognition using phase-based image matching. IEEE Trans. Pattern Anal. Mach. Intell. **30**(10), 1741–1756 (2008)
37. K. Takita, M. Muquit, T. Aoki, T. Higuchi, K. Kobayashi, A sub-pixel correspondence search technique for computer vision applications. IEICE Trans. Fundam. **E87-A**(8), 1913–1923 (2004)

Index

H.M. Waidyasooriya et al., *Design of FPGA-Based Computing Systems with OpenCL*, https://doi.org/10.1007/978-3-319-68161-0