

Programming BlueField DPU using ODOS

OpenMP DPU Offloading Support

Muhammad Usman, Sergio Iserte, Antonio J. Peña

Contents

1	Administrivia	3
1.1	Learning Outcomes	3
1.2	Resources	3
2	Introduction	4
2.1	What is BlueField DPU?	4
2.2	What is OpenMP Target Offloading?	4
2.3	How does OpenMP Target Offloading for BlueField Work?	4
3	Building ODOS-enabled LLVM	6
3.1	Setup LLVM Build	6
3.2	Compile LLVM Build	6
3.3	Setup OpenMP Build	6
3.4	Compile OpenMP Build	7
3.5	Check BlueField LLVM/OpenMP Target Info	7
4	Building and Running	
	ODOS-enabled Applications	8
4.1	Loading modules	8
4.1.1	Modules on Host Side	8
4.1.2	Modules on DPU Side	8
4.2	Compiling	8
4.3	Running	8
4.3.1	DOCA OpenMP Service in DPU	8
4.3.2	Application in Host	8

5	Labs	9
5.1	Setup	9
5.1.1	Shells	9
5.1.2	Node Allocation	9
5.1.3	Accessing Nodes	9
5.1.4	Loading Modules	9
5.1.5	Compile	9
5.1.6	Run	9
5.1.7	Clean	10
5.2	Task A: Hello World	10
5.3	Task B: OpenMP with Target Code	11
5.4	Task C: Shared Libraries	12
5.5	Task D: BlueField as a Network Accelerator	14
6	MPI Compatibility	21
7	Access ODOS	22
8	Contact Us	22

1 Administrivia

1.1 Learning Outcomes

By the end of this lab you will be able to;

1. Understand capabilities of OpenMP for programming BlueField DPUs.
2. Program BlueField DPUs for domain-specific applications.

1.2 Resources

- LLVM with OpenMP offloading support for BlueField
- Access to cluster with BlueField DPU devices

2 Introduction

2.1 What is BlueField DPU?

The BlueField Data Processing Unit (DPU) by NVIDIA is a versatile hardware platform designed for data center infrastructure. Combining high-performance networking and security acceleration, BlueField DPUs are frequently deployed as Smart Network Interface Cards (SmartNICs), known for their capacity to offload and expedite various data center workloads, particularly networking and security tasks, thereby lightening the load on the host CPU. Their presence in data centers contributes to improved operational efficiency and performance, enhancing overall data processing capabilities and security.

2.2 What is OpenMP Target Offloading?

OpenMP target offloading is a powerful feature that extends the capabilities of the OpenMP parallel programming model to include offloading computations to accelerators like GPUs or other coprocessors. With OpenMP target offloading, developers can leverage parallelism on both the CPU and accelerator devices, thereby enhancing the performance and efficiency of compute-intensive applications. This feature enables the seamless migration of compute-intensive tasks to accelerators while maintaining a unified code-base, making it a valuable tool for applications requiring high computational power and speed.

2.3 How does OpenMP Target Offloading for BlueField Work?

This DOCA-based ODOS solution is composed of three main modules:

- Cross-compilation: LLVM feature to generate a binary that can run in the DPU device.
- OpenMP BlueField plugin: Runtime for interacting with BlueField.
- OpenMP DOCA service: Runs on BlueField DPUs to accept and complete requests received from the host, through the OpenMP plugin.

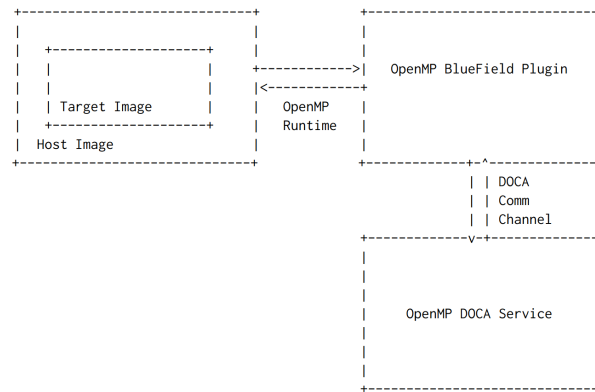


Figure 1: Methodology of OpenMP Target Offloading for BlueField DPU devices

A fat binary with embedded cross-compile code for BlueField (ARM architecture) is provided to OpenMP plugin by OpenMP runtime. The runtime then uses DOCA Comm Channel module to communicate with the DOCA OpenMP Service running on DPU. The service receives requests and run the tasks as required by the runtime through plugin. These tasks include loading image, allocation and copying of data and running target codes.

3 Building ODOS-enabled LLVM

Precompiled binaries will be shared in cluster. The following instructions are included herewith as a reference for compilation later.

3.1 Setup LLVM Build

Setup path for build directory and installation directory

```
1 $ export PREFIX=$HOME/opt/llvm_bf/  
2 $ export BUILDDIR=$HOME/tmp/build_llvm_bf/
```

Setup the build system using **cmake** command

```
1 $ cmake -S llvm-project/llvm  
2         -B llvm_objdir  
3         -DCMAKE_INSTALL_PREFIX="$PREFIX"  
4         -DCMAKE_BUILD_TYPE=Release  
5         -DCMAKE_C_COMPILER="which gcc"  
6         -DCMAKE_CXX_COMPILER="which g++"  
7         -DCMAKE_EXE_LINKER_FLAGS="$LDFLAGS"  
8         -DLLVM_BUILD_UTILS=OFF  
9         -DLLVM_ENABLE_PROJECTS="clang"  
10        -DGCC_INSTALL_PREFIX="/usr"  
11        -DCLANG_ENABLE_ARCMT=OFF  
12        -DCLANG_ENABLE_STATIC_ANALYZER=OFF
```

Note: the above command requires **ninja** to build LLVM

3.2 Compile LLVM Build

Compile using **ninja** build system

```
1 $ ninja -C $BUILDDIR install
```

3.3 Setup OpenMP Build

Setup path for build directory and installation directory

```
1 $ export PREFIX=$HOME/opt/llvm_bf/  
2 $ export BUILDDIR=$HOME/tmp/build_llvm_openmp_bf/
```

Setup the build system using **cmake** command:

```

1 $ cmake -S llvm-project/openmp
2       -B openmp_objdir
3       -DLLVMROOT="$PREFIX"
4       -DCMAKE_INSTALL_PREFIX="$PREFIX"
5       -DCMAKE_BUILD_TYPE=Release
6       -DCMAKE_C_COMPILER="$PREFIX/bin/clang"
7       -DCMAKE_CXX_COMPILER="$PREFIX/bin/clang++"
8       -DCMAKE_EXE_LINKER_FLAGS="$LD_FLAGS"
9       -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
10      -DOPENMP_ENABLE_LIBOMPTARGET_PROFILING=OFF
11      -DLIBOMP_HAVE_OMPT_SUPPORT=OFF
12      -DLIBOMP_INSTALL_ALIASES=OFF
13      -DLIBOMPTARGET_ENABLE_DEBUG=OFF
14      -DLLVM_BUILD_TOOLS=ON

```

3.4 Compile OpenMP Build

Compile using `ninja` build system

```

1 $ ninja -C $BUILDDIR install

```

3.5 Check BlueField LLVM/OpenMP Target Info

We can check if BlueField is enabled in our installation of LLVM/OpenMP by using the following command:

```

1 $ llvm-omp-device-info
2 ...
3 Device (4):
4     BlueField DPU device
5         iface      : ib0
6         ib dev     : mlx5_2
7         doca id    : 2
8         pci addr   : 42:00:0
9         comm channel:
10             max msg size      : 4080
11             max send queue size : 8192
12             max receive queue size : 8192
13 ...

```

4 Building and Running ODOS-enabled Applications

4.1 Loading modules

4.1.1 Modules on Host Side

```
1 $ module load ODOS
```

4.1.2 Modules on DPU Side

```
1 $ module load ODOS
```

4.2 Compiling

```
1 $ clang -fopenmp -fopenmp-targets=aarch64-unknown-linux \
2     app.c -o app
```

4.3 Running

4.3.1 DOCA OpenMP Service in DPU

On the DPU side, run the DOCA OpenMP service:

```
1 $ doca-omp-service
```

4.3.2 Application in Host

On host side, run the compiled code

```
1 $ ./app
```

Note: make sure that DOCA OpenMP service is running on the DPU.

5 Labs

5.1 Setup

5.1.1 Shells

Open up two terminals to access both host and DPU. Access HPC Advisory Council Clusters login node on both.

```
1 $ ssh username@gw.hpcadvisorycouncil.com
```

5.1.2 Node Allocation

```
1 $ salloc -N 2 -p thor --time 03:00:00 -w thor0XX,thorbf3aXX
```

5.1.3 Accessing Nodes

Access host node

```
1 $ ssh thor0XX
```

and access DPU node

```
1 $ ssh thorbf3a0XX
```

5.1.4 Loading Modules

On both sides, load the ODOS module and cmake

```
1 $ module load cmake ODOS
```

5.1.5 Compile

A script exists to compile all the labs.

```
1 $ sh compile.sh
```

5.1.6 Run

Run the DOCA OpenMP Service on DPU side

```
1 $ doca-omp-service
```

Run the application on host side

```
1 $ ./task_a/build/hello
```

Name of the binary generated can be different based on the task.
Moreover, task c and d require more changes and will be discussed in the relevant section.

5.1.7 Clean

```
1 $ sh compile.sh
```

5.2 Task A: Hello World

The task offloads code block to BlueField DPU. The example also demonstrates that DPU is capable of running linux systemcalls.


Code Snippet

```
1 #include <omp.h>
2 #include <stdio.h>
3
4 int main()
5 {
6 #pragma omp target
7     puts("Hi Folks!");
8
9     return 0;
10 }
```

Compilation Script

```
1 $ $LLVM/bin/clang \
2     -fopenmp -fopenmp-targets=aarch64-unknown-linux \
3     hello.c -o hello
```

Output Shot



```
uthmanhere@thor013:~/omp_exp/labs_sc23/task_a/build$ ./hello
uthmanhere@thor013:~/omp_exp/labs_sc23/task_a/build$

uthmanhere@thorbf3a013:~$ ./doca_openmp_service
Hi Folks!
```

5.3 Task B: OpenMP with Target Code

The task demonstrates that offloaded code can also use OpenMP features and hence make use of much more features of underlying hardware inside DPU.

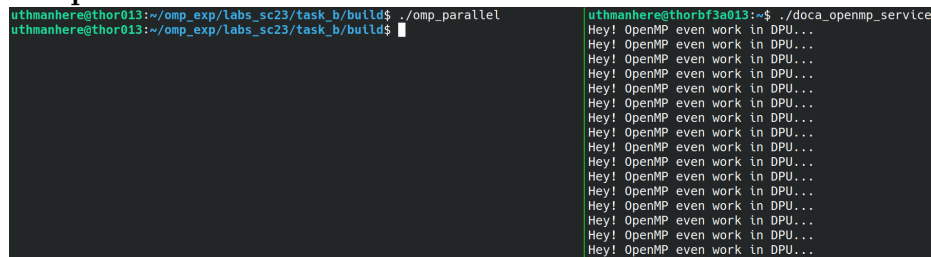
Code Snippet

```
1 #include <omp.h>
2 #include <stdio.h>
3
4 int main()
5 {
6 #pragma omp target
7 #pragma omp parallel
8     puts("Hey! OpenMP even work in DPU...");
9
10     return 0;
11 }
```

Compilation Script

```
1 $ $LLVM/bin/clang \
2     -fopenmp -fopenmp-targets=aarch64-unknown-linux \
3     omp_parallel.c -o omp_parallel
```

Output Shot



```
uthmanhere@thor013:~/omp_exp/labs_sc23/task_b/build$ ./omp_parallel
uthmanhere@thor013:~/omp_exp/labs_sc23/task_b/build$
uthmanhere@thorbf3a013:~$ ./doca_openmp_service
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
Hey! OpenMP even work in DPU...
```

5.4 Task C: Shared Libraries

The task demonstrates the use of shared libraries with BlueField DPU. It consists of multiple files as:

- Library header file - `log.h`
- Library source code file - `log.c`
- Application source code - `shared.c`

We generate 3 files from these files to be able to use them with LLVM/OpenMP Target Offloading infrastructure:

- Shared object - `liblog_x86.so`
- Cross-compile shared object - `liblog_aarch64.so`
- Application binary - `shared`

Setup Library Path Setup the path of the library that has been compiled on both host and dpu sides

```
1 $ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/task_c/build/
```

Source Code Library header file `log.h`

```
1 void printify();
```

Source Code Library source code file `log.c`

```
1 #include "log.h"
2
3 #include <stdio.h>
4
5 void printify()
6 {
7     puts("Hello from the other side (shared object actually)");
8 }
```

Compilation script Script for native and cross-compiled shared objects

```

1 $ LLVM/bin/clang log.c -shared \
2     -o liblog_x86.so
3 $ LLVM/bin/clang log.c -shared \
4     -target aarch64-unknown-linux \
5     -o liblog_aarch64.so

```

Source Code Application

```

1 #include <omp.h>
2 #include "log.h"
3 #include <stdio.h>
4
5 int main()
6 {
7     #pragma omp target
8         printf();
9
10     return 0;
11 }

```

Compilation script for application code

```

1 $ LLVM/bin/clang -fopenmp \
2     -fopenmp-targets=aarch64-unknown-linux \
3     shared.c -o shared \
4     -L. -llog_x86 \
5     -Wl,--device-linker=L,--device-linker=llog_aarch64

```

Output Shot

```

uthmanhere@thor013:~/omp_exp/labs_sc23/task_c/build$ ls
liblog_aarch64.so  liblog_x86.so  shared
uthmanhere@thor013:~/omp_exp/labs_sc23/task_c/build$ export \
> LD_LIBRARY_PATH="pwd":$LD_LIBRARY_PATH
uthmanhere@thor013:~/omp_exp/labs_sc23/task_c/build$ ./shared
uthmanhere@thor013:~/omp_exp/labs_sc23/task_c/build$

uthmanhere@thorbf3a013:~$ export LD_LIBRARY_PATH=\
> /global/home/users/uthmanhere/omp_exp/labs_sc23/task_c/build:\
> $LD_LIBRARY_PATH
uthmanhere@thorbf3a013:~$ ./doca_openmp_service
Hello from the other side (shared object actually..)

```

5.5 Task D: BlueField as a Network Accelerator

This task demonstrate capabilities of BlueField DPU as a network preprocessor or network post processor. It employs a TCP/IP application with performs an asynchronous pingpong with increment on DPU, which CPU is free to run other compute-centric workloads.

Source Code Main snippet contains a target block for DPU offloading. The target block contains initialization of a TCP/IP network. Then it proceeds to run a ping pong application to run asynchronous transfer and compute capabilities of the code.

```

1 int main(int argc, char *argv[])
2 {
3     int fd;
4     char mode;
5
6     if (argc < 2) {
7         mode = 's';
8     } else if (argc > 1 && argv[1][0] == 'c') {
9         mode = 'c';
10    } else {
11        err(-EINVAL,
12            "invalid argument." \
13            " Leave empty for server. c for client");
14        return -EINVAL;
15    }
16
17 #pragma omp target nowait
18 {
19     fd = tcp_init(mode);
20
21     if (fd < 0) {
22         err(-EAGAIN, "init failed.");
23         goto err_out;
24     }
25
26     puts("connection established.");
27
28     ping-pong(fd, mode);
29
30     close(fd);
31 err_out:
32 }

```

```
33
34 #pragma omp taskwait
35
36     return 0;
37 }
```

Source code Pingpong snippet first sends from client to server. The server increments the messages and sends it back the client. The client does the same i.e. increment the integer again and send it back to to server. It continues for pre-specified number of iterations.

```
1 void ping_pong(int fd, char mode)
2 {
3     int i, m;
4     m = 0;
5
6     for (i = 0; i < _ITERATIONS_; ++i) {
7         if (mode == 'c') {
8             printf("to server > %02d\n", m);
9             send(fd, &m, sizeof(m), 0);
10            recv(fd, &m, sizeof(m), 0);
11            ++m;
12        } else {
13            recv(fd, &m, sizeof(m), 0);
14            ++m;
15            printf("to client > %02d\n", m);
16            send(fd, &m, sizeof(m), 0);
17        }
18    }
19 }
20 }
```

Running

- Setup 4 nodes: 2 hosts and their 2 corresponding DPUs
- Arbitrarily choose one pair as server and the other pair as client
- Check the IP address of server pair

```
1 $ ip -c -br a
2 lo      UNKNOWN 127.0.0.1/8  ::1/128
3 eth0    UP 192.168.3.230/19 fe80::966d:aef:fe4f:4e3a/64
```

```

4 eth1      DOWN
5 ib0_mlx5  UP   192.168.131.230/19 fe80::966d:ae03:4f:4e26/64
6 ib1_mlx5  DOWN
7

```

The IPv4 address being 192.168.3.230 of interface `eth0`

- Replace the IP address in the code file `task_c/net_accel.c`

```

1 #define _SERVER_IP_ "192.168.3.230"

```

- Compile using `thcompile.sh` script
- Run `doca-omp-service` on both DPUs
- On server-side host, run:

```

1 ./net_accel
2

```

- On client-side host, run:

```

1 ./net_accel c
2

```

Output Shot

<pre> uthmanhere@thor013:~/omp_exp/labs_sc23/task_d/build\$./net_accel uthmanhere@thor013:~/omp_exp/labs_sc23/task_d/build\$ </pre>	<pre> uthmanhere@thorbf3a013:~\$./doca_openmp_service connection established. to client > 01 to client > 03 to client > 05 to client > 07 to client > 09 to client > 11 to client > 13 to client > 15 to client > 17 to client > 19 </pre>
<pre> uthmanhere@thor014:~/omp_exp/labs_sc23/task_d/build\$./net_accel c uthmanhere@thor014:~/omp_exp/labs_sc23/task_d/build\$ </pre>	<pre> uthmanhere@thorbf3a014:~\$./doca_openmp_service connection established. to server > 00 to server > 02 to server > 04 to server > 06 to server > 08 to server > 10 to server > 12 to server > 14 to server > 16 to server > 18 </pre>

Complete Source Code

```
1 #include <err.h>
2 #include <errno.h>
3
4 #include <omp.h>
5 #include <stdio.h>
6 #include <unistd.h>
7
8 #include <sys/types.h>
9 #include <sys/socket.h>
10
11 #include <arpa/inet.h>
12
13 #define _PORT_ 3310
14 #define _SERVER_IP_ "192.168.3.213"
15 #define _ITERATIONS_ (10)
16
17 int tcp_init(char MODE)
18 {
19     int fd, ret;
20     int enable_reuse = 1;
21     socklen_t socklen_;
22     struct sockaddr_in sockaddr_;
23     fd = socket(AF_INET, SOCK_STREAM, 0);
24     if (fd == -1) {
25         err(-EAGAIN, "failed to crate socket.");
26         return -EAGAIN;
27     }
28     ret = setsockopt(fd, \
29                     SOL_SOCKET, SO_REUSEADDR, \
30                     &enable_reuse, sizeof(enable_reuse));
31     if (ret == -1) {
32         err(-EAGAIN, "failed to set socket to use.");
33         return -EAGAIN;
34     }
35
36     sockaddr_.sin_family = AF_INET;
37     sockaddr_.sin_port = htons(_PORT_);
38
39     if (MODE == 'c') {
40         sockaddr_.sin_addr.s_addr = \
41             inet_addr(_SERVER_IP_);
42         ret = connect(fd, \
43                     (struct sockaddr *)&sockaddr_, \
44                     \
```

```

45         sizeof(sockaddr_));
46     if (ret == -1) {
47         err(-EAGAIN, \
48             "failed to connect socket.");
49         return -EAGAIN;
50     }
51 } else {
52
53     sockaddr_.sin_addr.s_addr = INADDR_ANY;
54
55     ret = bind(fd, \
56         (struct sockaddr *) &sockaddr_, \
57         sizeof(sockaddr_));
58     if (ret == -1) {
59         err(-EAGAIN, "failed to bind socket.");
60         return -EAGAIN;
61     }
62
63     ret = listen(fd, 1);
64     if (ret == -1) {
65         err(-EAGAIN, \
66             "failed to listen socket.");
67         return -EAGAIN;
68     }
69
70     socklen_ = sizeof(sockaddr_);
71     fd = accept(fd, \
72         (struct sockaddr *)&sockaddr_, \
73         &socklen_);
74     if (fd == -1) {
75         err(-EAGAIN, \
76             "failed to listen socket.");
77         return -EAGAIN;
78     }
79 }
80
81     return fd;
82 }
83
84 void ping_pong(int fd, char mode)
85 {
86     int i, m;
87     m = 0;
88
89     for (i = 0; i < ITERATIONS_; ++i) {

```

```

90         if (mode == 'c') {
91             printf("to server > %02d\n", m);
92             send(fd, &m, sizeof(m), 0);
93             recv(fd, &m, sizeof(m), 0);
94             ++m;
95         } else {
96             recv(fd, &m, sizeof(m), 0);
97             ++m;
98             printf("to client > %02d\n", m);
99             send(fd, &m, sizeof(m), 0);
100         }
101     }
102 }
103 }
104
105 int main(int argc, char *argv[])
106 {
107     int fd;
108     char mode;
109
110     if (argc < 2) {
111         mode = 's';
112     } else if (argc > 1 && argv[1][0] == 'c') {
113         mode = 'c';
114     } else {
115         err(-EINVAL,
116            "invalid argument." \
117            " Leave empty for server. c for client");
118         return -EINVAL;
119     }
120
121 #pragma omp target nowait
122 {
123     fd = tcp_init(mode);
124
125     if (fd < 0) {
126         err(-EAGAIN, "init failed.");
127         goto err_out;
128     }
129
130     puts("connection established.");
131
132     ping_pong(fd, mode);
133
134     close(fd);

```

```
135 err_out :  
136 }  
137  
138 #pragma omp taskwait  
139  
140     return 0;  
141 }
```

6 MPI Compatibility

We are actively working on MPI compatibility. **Coming soon...**

```
1 #include <stdio.h>
2 #include <mpi.h>
3
4 int main(int argc, char** argv) {
5     MPI_Init(&argc, &argv);
6
7     int rank; // Process rank
8     int size; // Total number of processes
9
10    MPI_Comm_rank(MPLCOMM_WORLD, &rank);
11    MPI_Comm_size(MPLCOMM_WORLD, &size);
12
13    if (size < 2) {
14        printf("requires at least two processes.\n");
15        MPI_Finalize();
16        return 1;
17    }
18
19    #pragma omp target
20    {
21        int data = 0;
22
23        if (rank == 0) {
24            data = 42; // Data to be sent from rank 0
25            MPI_Send(&data, 1, MPI_INT, \
26                    1, 0, MPLCOMM_WORLD);
27            printf("Process %d sent data: %d\n", rank, data);
28        } else if (rank == 1) {
29            MPI_Recv(&data, 1, MPI_INT, \
30                    0, 0, MPLCOMM_WORLD, MPI_STATUS_IGNORE);
31            printf("Process %d received data: %d\n", rank, data);
32        }
33    }
34    MPI_Finalize();
35
36    return 0;
37 }
```

7 Access ODOS

Get access to the ODOS framework in AccelCom website:

[bsc.es/discover-bsc/organisation/scientific-structure/
accelerators-and-communications-hpc/team-software](https://bsc.es/discover-bsc/organisation/scientific-structure/accelerators-and-communications-hpc/team-software)

8 Contact Us

Don't hesitate contacting us at accelcom@bsc.es for further information.

