NFVRG
Internet-Draft
Intended Status: Informational

Bose Perumal
Wenjing Chu
R. Krishnan
Hemalathaa. S
Dell

Expires: December 3 2015
June 29 2015

# NFV Compute Acceleration Evaluation and APIs
# draft-perumal-nfvrg-nfv-compute-acceleration-00

## Abstract

Network functions are being virtualized and moved to industry standard servers. Steady growth of traffic volume requires more compute power to process the network functions. Network packet based architecture provides a lot of scope for parallel processing. Generic parallel processing can be done in common multicore platforms like GPUs, coprocessors like Intel Xeon Phi[6][7] and Intel[7]/AMD[10] multicore CPUs. In this draft to check the feasibility and to exploit this parallel processing capability, multi string matching is taken as the sample network function for URL filtering. Aho-Corasick algorithm has been made use for multi pattern matching. Implementation utilizes OpenCL [3] to support many common platforms[7][10][11]. A list of optimizations is done, the application is tested on Nvidia Tesla K10 GPUs. A common API for NFV Compute Acceleration has been proposed.

## Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/1id-abstracts.html

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html



Table of Contents

# 1 Introduction

Network equipment vendors use specialized hardware to process data at a low latency and high throughput. Packet processing above 4 Gb/s is done using expensive, purpose-built application-specific integrated circuits. However, the low unit volumes force manufacturers to price these devices at many times the cost of producing them, to recover the R&D cost.

Network Function Virtualization (NFV)[1] is a key emerging area for network operators, hardware and software vendors, cloud service providers, and in general network practitioners and researchers. NFV introduces virtualization technologies into the core network to create a more intelligent, more agile service infrastructure. Network functions that are traditionally implemented in dedicated hardware appliances will need to be decomposed and executed in virtual machines running in data centers. The parallelism of graphics processor provides it the potential to function as network coprocessor.

Network virtual function is responsible for specific treatment of received packets. A network virtual function can act at various layers of a protocol stack. When there is more compute power, multiple virtual network functions can be executed in a single system or VM. When multiple virtual network functions are processed in a system, some of them could be processed in parallel with other network functions. This paper proposes a method to represent ordered set of virtual network functions in a combination of a sequential and parallel order. This draft is for software based network functions, so any further reference to network function means virtual network function.

Software written for specialized hardware like network processors, ASIC, FPGA, is closely tied to the hardware and specific vendor products. It cannot be reused in other hardware platforms. For generic compute acceleration different hardware platforms can be used, like GPUs from different vendors, Intel Xeon Phi coprocessors and multi core CPUs from different vendors. All these compute acceleration platforms support OpenCL as parallel programming language. Instead of every vendor writing OpenCL code, NFV Compute Acceleration (NCA) API has been proposed for a common compute accelerator in this paper. This API will be a library with C API functions for declaring the network functions as an ordered set and moving packets around.

Multi-pattern string matching is used in a number of applications including network intrusion detection and digital forensics. Hence multi pattern matching is chosen as a sample network function. Aho-

Corasick[2] algorithm with few modifications has been used to find the first occurrence of any pattern from the signature database. Based on this network function the throughput numbers are measured.

## 1.1 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

## 2. OpenCL based Virtual Network Function Architecture

Network functions like multi pattern matching is process intensive and common for multiple NFV applications. Generic compute acceleration framework functions and service specific functions are clearly separated in this prototype. The architecture diagram with one network function is shown in Figure 1. Multiple network functions can also be loaded. Most of the signature based algorithms like Aho-Corasick[2], Regex[8],etc. generate a Deterministic Finite Automaton(DFA)[2][8]. DFA database is generated in CPU and loaded to the accelerator. Kernels executed in the accelerator will use the DFA.

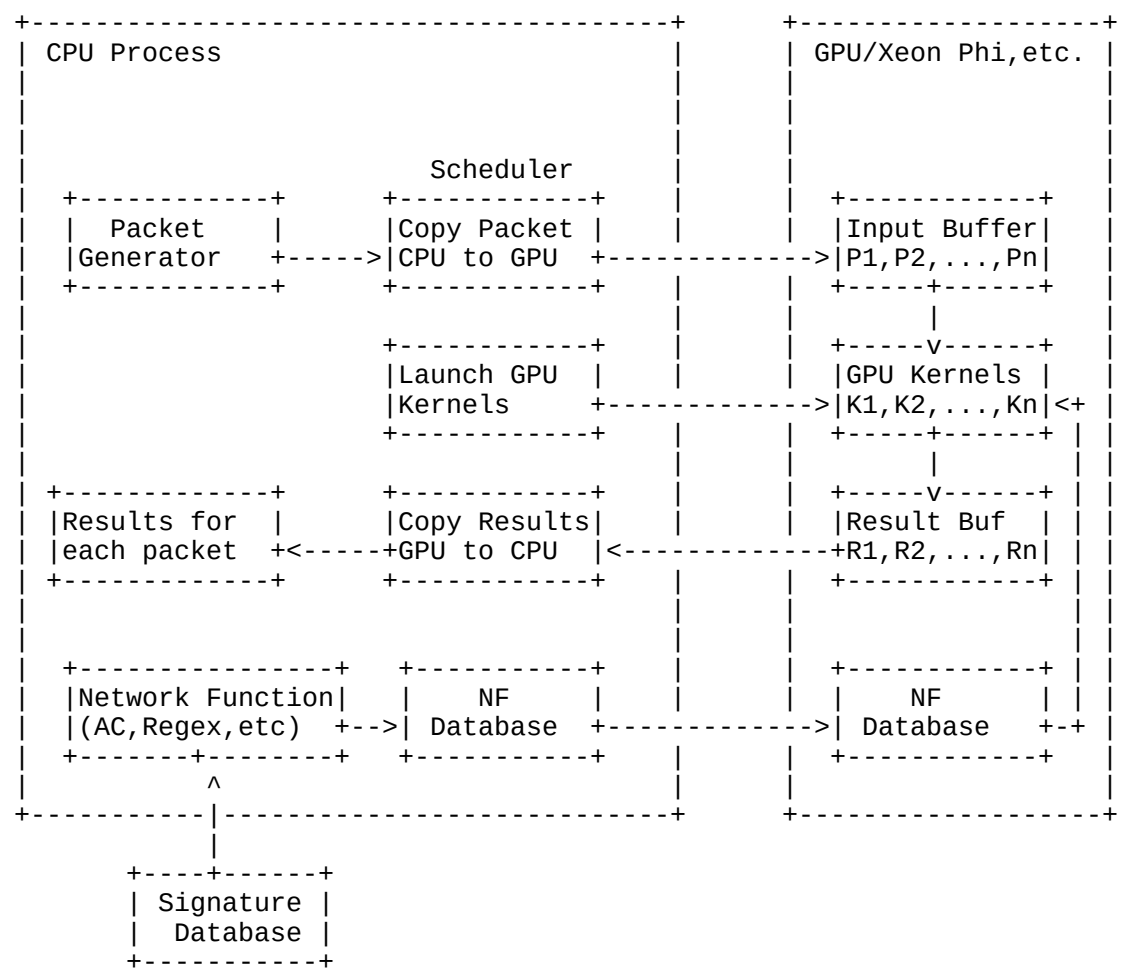


Figure 1. OpenCl based Virtual Network Function Software Architecture Diagram

## 2.1 CPU Process

Accelerators like GPUs or coprocessors will augment CPU and currently they cannot function alone. Network virtual function is split between CPU and GPU. CPU process owns the packet preprocessing, packet movement and scheduling. GPUs will do the core functionality of the network functions. CPU process interfaces between the packet I/O and GPU. During initialization it does set of following functions.

1. Device Discovery
2. Initialize OpenCL object model
3. Initialize memory module
4. Initialize network functions
5. Trigger scheduler

## 2.2 Device Discovery

Using OpenCL functions device discovery module discovers the platforms and devices. Based on number of devices discovered, device context and command queues are created.

## 2.3 Mixed Version Support

OpenCl is designed to support devices with different capabilities under a single platform[3]. There are three version identifiers in OpenCl, the platform version, the version of a device, and the version(s) of the OpenCl C language supported on a device.

The platform version indicates the version of the OpenCL runtime supported. The device version is an indication of the devices capabilities. The language version for a device represents the OpenCL programming language features a developer can assume are supported on a given device.

OpenCl C is designed to be backwards compatible, so a device is not required to support more than a single language version to be considered conformant. If multiple language versions are supported, the compiler defaults to using the highest language version supported for the device.

Code written for old device version may not utilize the full capabilities of new device if there are hardware architectural changes.

## 2.4 Scheduler

Scheduling between the packet buffers coming from the network I/O to the device command queues is carried out by the scheduler. Scheduler operates on following parameters.

- N - Number of Packet buffers (Default 6)
- M - Number of Packets in each buffer (Default 16384)
- K - Number of Devices (Discovered 2)
- J - Number of Command Queues for each device (Default 3)
- I - Number of Commands to the device to complete single network function (Default 3)
- S - Number of network functions executed in parallel. (Default 1)

Default values mentioned above are for the best results in our current hardware environment and multi string match function.

Operations for completing network function for one packet buffer

1. Identify a free command queue
2. Copy packets from IO memory to pinned memory for GPU
3. Fill Kernel function parameters
4. Copy pinned memory to GPU global memory
5. Launch kernels for number of packets in the packet buffer
6. Check kernel execution completion and collect results
7. Report results to application

Scheduler calls OpenCl API with number of kernels to be executed in parallel. Distributing the kernels to cores is taken care by OpenCl library. If there are any error during launching the kernels, OpenCl API returns error and appropriate error handling can he done.

## 3. Aho-Corasick Algorithm

The Aho-Corasick algorithm [2] is the most effective multi pattern matching algorithm. Aho-Corasick algorithm is a kind of dictionary-matching algorithm that locates elements of a finite set of strings within an input text. The complexity of the algorithm is linear in the length of the patterns plus the length of the searched text plus the number of output matches.

The algorithm works in two parts. The first part is the building of the tree from keywords that needs to be searched for, and the second part is searching the text for the keywords using the previously built tree (state machine). Searching for a keyword is efficient, because it only moves through the states in the state machine. If a character is match, goto () function is executed otherwise it follows fail () function. Match found is returned by the out () function.

All the three functions just access the indexed data structures and return the value. goto () data structure is a two dimension matrix accessed based on current state and currently compared character. fail () function is an array, which has the link to alternate path for each state. Out function is an array of states and it has the records on whether the string search has completed on a particular state.

Based on the signature database, all three data structures are constructed in CPU. These data structures are copied to GPU global memory during the initialization stage. Pointers to these data structures are passed as the kernel parameter when the kernels are launched.

## 4. Optimizations

For this prototype Nvidia Tesla K10 GPU [5] is used which has 2 processors with 1536 cores each running at 745MHz. Each processor has 4GB of memory attached to it. It is connected to CPU via PCI 3.0 x16 interface.

Server used is Dell R720 which has Intel Xeon 2665 with 2 processors each having 16 cores. Only one CPU core is used for our experiment.

### 4.1 Variable size packet packing

Multiple copies from CPU to GPU is costly. Packets are batched for processing in GPU. Packet sizes vary from 64 bytes to 1500 bytes. Having a fixed size buffer for each packet, leads to copying a lot of unwanted memory from CPU to GPU in case of smaller number of packets.

For variable size packing one single large buffer is allocated for number of packets in the batch. Initial portion of the buffer has the packet start offsets for all packets. At the packet offset, packet size and packet contents are placed. Only buffer size filled with packets is copied from CPU to GPU.

### 4.2 Pinned Memory

Memory allocated using malloc is paged memory. When coping from CPU to GPU, memory is first copied from paged memory to non-paged memory, then it is copied from non-paged memory to GPU global memory.

OpenCL provides commands and procedure to allocate and copy memory from non-paged memory[3][4]. Using this pinned memory avoids one internal copy and showed 3x improvements in memory copy time. In our experiments pinned memory was used for CPU to GPU packet buffer copy and GPU to CPU result buffer copy.

### 4.3 Pipelined Scheduler

OpenCL supports multiple command queues and Nvidia supports 32 command queues. Using non-blocking calls, commands can be placed on each queue. When GPU kernel functions are being executed, memory copy between CPU and GPU can happen in parallel.

In our experiment 6 command queues were created, 3 queues for each GPU processor. Copy packet buffer to GPU, Launch GPU kernel functions and read results from GPU are executed in parallel for 6 batches of data. Scheduling is performed using round robin to maintain packet order. Using pipe lining, allows hiding 99% of copy time and utilizing the full processing power of GPU.

### 4.4 Reduce Global memory access

OpenCL architecture and NVidia GPU architecture has 3 levels of memory, Global, Local and Private. Packets from CPU are copied to GPU global memory. Global memory access is costly and Char by Char access is not efficient.

Accessing private memory is faster but private memory is small, it cannot hold complete packet. So packets are copied as 32 bytes at a time using vload8 and float type.

### 4.5 Organizing GPU cores

Number of kernels functions(global-size) lunched should be more than the number GPU cores to hide latency. GPU provides the sub-grouping of cores to share memory. Optimal grouping size(local-size) is

calculated specific to GPU card.

## 5. Results

Using Aho-Corasick algorithm measured the performance of GPU system with different parameters. Signature database is the top website names. Ethernet and IP headers are skipped in the search, which is 34 bytes for each packet. Only protocol header analysis or application header analysis can also be performed.

Aho-Corasick algorithm is modified to match any one string from the signature database. After the first string is matched, result is written in the result buffer and function exits. If none of the string matched in the packet, whole packet is searched, this is the worst-case performance. If any one of the string matched earlier then remaining packet is not searched.

To understand the performance and to keep track of the timing of how the commands execute, OpenCl supports a function clGetEventProfilingInfo, which allows querying cl_event to get counter values. The device time counter is returned in nanoseconds.

For these experiment results Nvidia Tesla K10 GPU and Dell R720 server is used.

Results were taken by executing on bare metal server on Linux. Same code can be executed inside the Virtual Machine also.

### 5.1 Worst case 0 string match

Measured performance by varying signature database size to 1000, 2500 and 5000. Fixed size packets were generated with packet sizes 64/583/1450. Variable size packet generated with packet sizes from 64 to 1500 with an average packet size of 583 bytes and the results are shown in Table 1 and Table 2.

| No of Strings | 64 Fixed | 583 Fixed | 1450 Fixed | 583 Variable |
|---|---|---|---|---|
| 1000 | 37.03 | 30.74 | 31.68 | 15.08 |
| 2500 | 37.03 | 30.17 | 31.15 | 14.94 |
| 5000 | 36.75 | 30.03 | 31.15 | 14.87 |

Table 1: Bandwidth in Gbps for different packet sizes of traffic

| No of Strings | 64 Fixed | 583 Fixed | 1450 Fixed | 583 Variable |
|---|---|---|---|---|
| 1000 | 77.67 | 7.07 | 2.93 | 3.47 |
| 2500 | 77.41 | 6.95 | 2.88 | 3.44 |
| 5000 | 77.07 | 6.91 | 2.88 | 3.42 |

Table 2: Number of packets in Million packets per second(mpps) for different packet sizes of traffic

Varying signature database size with 1000, 2500 and 5000 do not have any major impact. State machine size gets bigger based on signature database size, but processing time for the packets remain the same.

For fixed size packets total bandwidth processed was always above 30 Gbps. For variable bandwidth packets total bandwidth processed is 14.9 Gbps.

Variable packet sizes vary from 64 to 1500 bytes, each packet being assigned to one core. The core which finishes early is idle till other cores complete their work. So full GPU power is not effectively utilized when using variable length packets.

## 5.2 Packet match

Having match percentage as the key, different parameters are measured. Table 3 shows the match percentage against the bandwidth in Gbps. For this experiment variable size packets with an average of 583 bytes are used. 16384 packets are batched for processing in GPU and 16384 threads are instantiated. Each packet is checked for 5000 strings.

| % of packets matched | Bandwidth in Gbps |
|---|---|
| 0 | 14.87 |
| 15 | 18.50 |
| 25 | 20.85 |
| 35 | 33.02 |

Table 3: Bandwidth in Gbps for different packet match percentage

| % of packets matched | No of Packets in mpps |
|---|---|
| 0 | 3.42 |
| 15 | 4.25 |
| 25 | 4.80 |
| 35 | 7.60 |

Table 4: Number of packets in Mpps for different packet match percentage

The packet match percentage against number of packets processed in mpps is shown in Table 4. Worst case experiment is 0 packets matched, so whole packet need to be searched. Time for single buffer(16384 packets) copy from CPU to GPU is 0.903 milliseconds. Kernel execution time for single buffer is 9.784 milliseconds. Result buffer copy from GPU to CPU is 0.161 milliseconds. Total of 209 buffers processed in one second, which is 3.42 million packets and 14.9 Gbps.

Best case experiment was executed with 35% of packets match. Time for single buffer(16384 packets) copy from CPU to GPU is 0.923 milliseconds. Kernel execution time for single buffer is 4.38 milliseconds. Result buffer copy from GPU to CPU is 0.168 milliseconds. Total of 464 buffers processed in one second, which is 7.6 million packets and 33.02 Gbps.

## 6. Compute Acceleration API

Multiple compute accelerators like GPUs, Coprocessors, ASICs/FPGAs and multi core CPUs can be used for NFV. Having a common API for NFV Compute Acceleration (NCA) can abstract the hardware details and enable NFV applications to use compute acceleration. This API will be a C library, user can compile it along with their code.

The delivery of end-to-end services often requires various network functions. Compute acceleration APIs should support the definition of ordered set of network functions and subset of these network functions which can be processed in parallel.

### 6.1 Add Network Function

Multiple network functions can be defined in the system. Network functions are identified by network function id. Based on service chain requirement network functions are dynamically loaded to the cores and executed. The API function nca_add_ network-function adds a new network function to the NCA.

In OpenCL terminology kernel is a function or set of function executed in compute core. OpenCL code files are small files with these functions called kernel functions.

```
int nca_add_network_function(
  int network_func_id,
  int (*network_func_init)(int network_func_id,void *init_params),
  char *cl_file_name,
  char *kernel_name,
  int (*set_kernel_arg)(int network_func_id,void *sf_args,
                 char *pkt_buf),
  int result_buffer_size
)

network_func_id    : Network function identifier unique
                     for every network function in the framework
network_func_init  : Initializing the network function, with the
                     device memory allocations, service
                     specific data structures are created.
cl_file_name       : File with network function kernel code
kernel_name        : Network function kernel entry function name
set_kernel_arg     : Function that will setup kernel arguments
                     before calling the kernel
result_buffer_size : result buffer size for this service function
```

### 6.2 Add Traffic Stream

Traffic streams are identified by stream id. Traffic streams are initialized with number of buffers and size of each buffer allocated for this stream. Each buffer is identified by a buffer id and it can hold N number of packets. These buffers are treated like ring buffers. These buffers are allocated as a contiguous memory by NCA and the pointer is returned.

Any notification during buffer processing is given through the callback function with stream_id, buffer_id and event.

Traffic stream is associated with a service function chain.

Service function chain is defined by three parameters. Number of network functions is mentioned in num_network_funcs. Actual network function ids are in service_function_chain array. Network functions are divided into subsets. Each subset has a subset number. Network functions within the subset can be executed in parallel. Subsets should be executed in sequence. There is a special subset number 0, which can be executed independent of any network functions in the chain

For example 6 service functions are represented below.

```
num_network_funcs        = 6;
service_func_chain       = {101, 105, 107, 108, 109, 110 }
network_func_ordered_set = {1, 1, 1, 2, 2, 0}
```

In the above example subset 1 which 101, 105, 107 should be executed first. Within this subset all 3 can be executed in parallel. After subset 1 subset 2 which is 108,109 will be executed. Subset 0 does not have any dependencies; scheduler can execute it at any time.

```
typedef struct dev_params_s {
         int dev_type,
         int num_devices,
} nca_dev;

int nca_traffic_stream_init (
  int num_buffers,
  int buffer_size,
  int (*notify_callback)(int buffer_id,int event)
  int num_network_funcs,
  int service_func_chain[CAF_MAX_SF],
  int network_func_parallel_sets[CAF_MAX_SF],
  nca_dev dev_params,
 )

stream-id          : Unique id to identify traffic stream
num_buffers        : Number of buffers
buffer_size        : Size of each buffer
notify_callback I  : Event notification callback.
num_service_funcs  : Number of service functions
                in the service chain
service_func_chain : Service function ids in this service chain
network_func_parallel_set : subsets for sequential and parallel
                   ordering of  service functions.
dev_params          : For this traffic stream user can choose the
                      device for  processing
Return Value       : stream-id which is unique to identify
                      traffic stream
```

### 6.3 Add Packets to Buffer

Packets are added to the buffer directly by the client application or by calling nca_add_packets. One or more packets can be added to the buffer.

```
int nca_add_packets(
  int context_id,
  int buffer_id,
  char * packet,
  int packet_len[]
  int num_packets
)

stream_id       : Steam id of the traffic stream
buffer_id       : Idenitfy the buffer to add the packet
packet          : Packet contents
packet_len[]    : Length of each packet
num_packets     : Number of packets
```

### 6.4 Process Packets

Once packets are filled in the buffer, nca_buffer_ready is called to process the buffer. This function can also be called without filling the complete buffer. NCA scheduler marks this buffer for processing.

```
int nca_buffer_ready(
  int context_id,
  int buffer_id
)
stream_id      :  Stream id identifies the traffic stream
buffer_id      :  Identify the buffer to add the packet
```

### 6.5 Event Notification

NCA will notify event about the buffer using the registered callback function. After the buffer is processed for the registered services, notify event callback is called. Client can read the result buffer.

```
int (*notify_callback) (
  int stream_id,
  int buffer_id,
  int event
)
stream_id      :  Stream id identifies the traffic stream
buffer_id      :  Identify the buffer to add the packet
event          :  Event maps to one of the buffer events. If
          the event is not specific to a buffer, buffer id is 0
```

### 6.6 Read Results

Client can read the results after service chain processing. Service chain processing completion is notified by an event through call back function.

```
int caf_read_results(
  int context_id,
  int buffer_id,
  char *result_buffer
)
stream_id      : Stream id identifies the traffic stream
buffer_id      : Identify the buffer to add the packet
result-buffer  : Result buffer pointer to be copied.
```

## 7. Other Accelerators

The prototype multi string search written in OpenCL successfully compiled and executed on both Intel Xeon Phi coprocessor and CPU only system with minimal changes in make file. For CPU only systems memory copies can be avoided. Since the optimizations for these platforms are not carried out, the performance numbers are not published.

## 8. Conclusion

To get best performance out of GPUs with large number of cores, number of threads executed in parallel should be large. For a single network function the latency will be in milliseconds, so it will be suited for network monitoring functions. If GPUs are tasked to do multiple network functions in parallel it can be used for other NFV functions.

Assigning single core for each packet gives best performance when all packet sizes are equal. For variable length packets performance goes down because the core processing the smaller packet has to be idle till the other cores complete processing the larger packets.

Code written in OpenCL is easily portable to other platforms like Intel Xeon Phi, multicore CPU with just make file changes. Though the same code execute correctly on all platforms, to achieve good performance, platform specific optimizations need to be done.

Proposed a network compute acceleration framework which will have all hardware specific optimizations and expose high level APIs to the applications. A set of APIs for defining traffic streams, network function addition and declaration of service chain with ordering method which include sequential and parallel.

## 9. Future Work

Dynamic device discovery and optimized code for different algorithms and devices will make NCA as a common platform to develop applications on top of this.

Integration of compute acceleration with IO acceleration technologies like Intel DPDK[9] can provide a complete networking platform for the applications.

Verification and performance measurement of compute acceleration platform running inside a Virtual Machines. Compute acceleration platform running inside Linux containers or Docker.

## 10 Security Considerations

Not Applicable

## 11 IANA Considerations

Not Applicable

## 12 References

### 12.1 Normative References

### 12.2 Informative References

[1] ETSI NFV White Paper: "Network Functions Virtualisation, An Introduction, Benefits, Enablers, Challenges, & Call for Action,"http://portal.etsi.org/NFV/NFV_White_Paper.pdf"

[2] A.V.Aho and M.J.Corasick, "Efficient string matching:An aid to A.v. Aho and M.j.Corasick, "Efficient string matching:An aid to bibliographic search",Communications of the ACM, vol. 20, Session 10.

[3] OpenCl Specification, "https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf"

[4] OpenCl Best Practices Guide, "http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf"

[5] Nvidia Tesla K10, http://www.nvidia.in/content/PDF/kepler/Tesla-K10-Board-Spec-BD-06280-001-v07.pdf

[6] Intel Xeon Phi "http://www.intel.in/content/www/in/en/processors/xeon/xeon-phi-detail.html"

[7] Intel OpenCl "https://software.intel.com/en-us/intel-opencl"

[8] Implementing Regular Expressions "https://swtch.com/~rsc/regexp/"
[9] Intel DPDK "http://dpdk.org/"
[10] AMD OpenCl Zone, "http://developer.amd.com/tools-and-sdks/opencl-zone/"
[11] Nvida OpenCl "https://developer.nvidia.com/opencl

## Acknowledgements

The authors would like to thank the following individuals for their support in verifying the prototype in different platforms : Shiva Katta and K. Narendra.

## Authors' Addresses

Bose Perumal
Dell
Bose_Perumal@Dell.com

Wenjing Chu
Dell
Wenjing_Chu@Dell.com

Ram (Ramki) Krishnan
Dell
Ramki_Krishnan@Dell.com

Hemalathaa S
Dell
Hemalathaa_S@Dell.com