

Blow up the CPU Chains!

OpenCL-assisted Network Protocols

Jens Heuschkel

TK / TU Darmstadt

heuschkel@tk.tu-darmstadt.de

Rick Vogel

TK / TU Darmstadt

vogel@tk.tu-darmstadt.de

Marcel Blöcher

DSP / TU Darmstadt

bloecher@dsp.tu-darmstadt.de

Max Mühlhäuser

TK / TU Darmstadt

max@tk.tu-darmstadt.de

Abstract—In times of cloud services and IoT, network communication takes heavily part of distributed applications. The underlying network protocol processing causes substantial CPU utilization, although recent kernel-bypass frameworks or specialized server network cards reduce the burden for the CPU. Since hardware acceleration is expensive and still not very common, the leverage of available compute power is a logical step, that leads to GPGPU utilization. The new heterogeneous uniform memory access architecture unites CPU and GPU on one die and therefore features a shared memory excluding the overhead of PCI-E communication.

In this paper we present *SocketCL*, a user-space network protocol stack acceleration. In order to utilize the GPU we ported CPU based Ethernet, IPv4 and UDP network protocols to OpenCL 2.0 kernels leveraging the new hUMA architecture. Resulting from minimal copy processes and optimal utilization of a system's compute resources, *SocketCL* achieves substantial acceleration of individual network protocols.

Index Terms—network protocol acceleration, GPGPU, heterogeneous uniform memory access

I. INTRODUCTION

Trends like IoT, smart speaker (e.g., Amazon Echo), and even smart-phones show that we depend on cloud services everywhere. Hence, one of the most important functions of our devices are network connections, and thus, the network protocol processing. Even though industry offers hardware acceleration for TCP as part of their high-end server network cards, for end-user, mobile or IoT devices, hardware acceleration is still not state-of-the art. In the last few years, general purpose computation on graphics processing units (GPGPU) become very popular to support various tasks even in end-user scenarios. This and the gaming industry lead to a big availability of computation power by GPUs. Since the network packet processing in high throughput scenarios follows the single instruction multiple data (SIMD) work load pattern, it is a well suited task for GPGPU. Despite the fact, ARM based devices gain a lot GPU power with every new generation (and therefore providing the basis for our approach) using GPU as network acceleration could be a flexible, update-able solution for data-centers too.

A major challenge of many similar software solutions was to compensate the overhead caused by the data transfer between the CPU and PCI/PCI-E connected GPUs. To address this overhead in general, a new heterogeneous system architecture (HSA) was released [15]. It locates the CPU and GPU on the same die and allows both to communicate through a

shared memory. A key difference in comparison to PCI/PCI-E connected GPUs is the uniform memory access in a HSA compliant system. Both the CPU and GPU share a common memory. Therefore, this kind of memory access is called heterogeneous uniform memory access (hUMA). Since this system architecture is still relatively young, the possible performance increase and scalability of a GPU accelerated packet processing solution is yet to be researched.

In order to examine these properties, we present *SocketCL*, a user-space network stack acceleration utilizing the GPU to accelerate the network protocol stack. We therefore focused on the send portion of an UDP, IPv4 and Ethernet protocol stack. Our implementation is based on OpenCL 2.0 specification, providing an interface to exploit shared virtual memory (SVM) functionalities provided by the hUMA system.

In order to provide comprehensible understanding of our solution, we discuss the hardware specifics together with the related work in Section II. In Section III we discuss our design decisions to realize a GPU based acceleration. Finally, in Section IV we show our measurement results and discuss them. The paper is concluded in Section V together with our ideas for future work.

II. BACKGROUND AND RELATED WORK

This section provides fundamental information on the differences between CPUs and GPUs by explaining hardware architectural properties and their impact on software models. It points out the key abstractions used within the OpenCL framework and also its correlation to a GPU's hardware architecture. Furthermore we present related work to this field.

Mukherjee et al. compares in [11] OpenCL 2.0 and HSA 1.0 by using OpenCL 1.2 as the baseline. In particular, it highlights new features of the OpenCL 2.0 specification such as shared virtual memory (SVM), dynamic parallelism, atomics as well as coarse-grained and fine-grained synchronization of SVM. An additional part of the article's work are multiple benchmarks that compare the performance of OpenCL 2.0 and HSA 1.0. Due to HSA 1.0's reduction of up to 92% in communication overhead, it outperforms OpenCL 2.0 in almost every benchmark. Beside the benchmarks the paper is a comprehensible introduction into OpenCL 2.0's new mechanisms and provides useful measurements on general tasks.

The paper [4] provides indispensable knowledge on GPUs, their hardware properties and strategies to utilize them advantageously. It explains why GPUs have numerous ALUs and how they achieve high data throughput rates by exploiting instruction level parallelism like SIMD. Not only does the paper highlight advantages of a GPU but also the challenges towards designing code that is optimized for the underlying hardware architecture. Furthermore, the paper explains a GPU's capability to hide memory latencies by its advanced thread scheduler.

In [5] Han et al. presents PacketShader, a software router framework using GPU's massive parallelism to overcome the CPU as a bottleneck. Since it uses a PCI-E connected GPU, one of the main challenges was to amortize the host to device memory copy overhead as well as kernel launch latencies. It also used multiple CPU threads to do packet input and output operations to utilize multiple NICs at once. In order to amortize the kernel launch costs, PacketShader collects multiple packets in a batch to process them in parallel on the GPU. A core difference to this Paper is that SocketCL leverages a hUMA system with a shared memory for CPU and GPU as well as our focus on end-nodes rather than routers.

As SocketCL replaces network packet processing, we need to bypass the kernel level protocol implementations. Therefore, we leveraged raw sockets [7]. Bypassing the kernel was already used in other user-level network stack implementations as [14], [10].

a) Memory Access Models: We distinguished between Uniform Memory Access (UMA), Non Uniform Memory Access (NUMA) and heterogeneous Uniform Memory Access (hUMA): In a UMA system all processing cores share a joint memory address space. As consequence, if two cores exchange an address, both cores can dereference this address and access the memory pointed at with equal access time to memory, but processor cores have to be identical [16]. The addition of a GPU, require an address translation between the CPU's and GPU's address space, hence forming a NUMA system. There is no guarantee for equal memory access times across the processors in a NUMA system [16]. Heterogeneous System Architecture (HSA) unifies the address space used by both the CPU and GPU and is therefore called hUMA [15]. It enables the CPU and GPU to dynamically allocate memory from the entire memory space.

b) GPU Thread Management: The hardware architecture of a GPU differs substantially from a CPU, allowing efficient and advantageous thread management. This efficiency is made possible by multiplying computing units and reducing their control logic. Therefore, a GPU is capable of launching and scheduling an enormous amount of lightweight threads with low context switching costs[4].

A AMD's GCN compliant GPU can have multiple compute units (cores). Each compute unit has four 16-lane wide SIMD units. Therefore, one compute unit can operate on 64 work-items concurrently. These 64 work-items can belong to different wavefronts since each SIMD unit has its own program counter [1]. Consequently, each SIMD unit can execute a

dissimilar kernel. Furthermore, each SIMD has an instruction buffer for 10 wavefronts that are being executed in parallel. According to the GCN architecture, assigning a wavefront to a SIMD unit takes one cycle, but the total execution for applying an operation onto all 64 work-items takes four cycles.

c) OpenCL: OpenCL is an application programming interface for developing parallel code targeting GPUs as well as CPUs. The execution model abstracts and differentiates between a host and its devices. A host can have multiple devices to which it schedules tasks to perform. The communication of tasks from the host to its devices is realized by using queues that can execute either in-order or out-of-order. The key abstractions towards these tasks are kernels, work-items, work-groups and wavefronts. A kernel can be seen as a piece of code executed in parallel across a set of work-items, each being an instance of that code. The execution of these work-items can be partitioned into multiple work-groups each being a set of work-items. A work-item can be compared to a single thread that belongs to a work-group of multiple work-items hence representing multiple threads. It is the smallest logical unit on the lowest level of OpenCL's execution model. A wavefront, on the other side, is a hardware specific partitioning of a work-group into a subset of work-items that can be executed concurrently on a compute unit. This abstraction is following OpenCL's SPMD model to fully utilize a GPU's massive hardware parallelism.

d) Memory Model: OpenCL abstracts and separates the physical memory into disjoint address spaces: global, local, private and constant [9]. The private address space is being used similarly to a stack in a CPU based program and is associated with a work item. Communication between work items of the same work group can be done through the local or global memory. Constant and global memory spaces are the biggest in size but the slowest as well. Because of that, communication is preferably done through local memory space.

OpenCL 2.0 introduced the term SVM, defining a conjoint address space. Thus, a host's pointer into SVM address space can be dereferenced by its devices and vice versa. In older versions of OpenCL, memory buffers had to be copied from the host to the device. Additionally, a buffer could be allocated as a mapped memory enabling direct memory access (DMA) for the device onto the host allocated memory, hence accelerating the copy process. SVM on the other hand enables the host and its devices to share memory without the need of copying it.

Figure 2 shows the OpenCL host, its OpenCL device as well as their associated address spaces. As explained above, the content of a regular non-SVM OpenCL buffer had to be copied from the host to the device and vice versa. Consequently, both the host and device had their own physical copy of that buffer, and thus, different addresses. A SVM buffer, on the other hand, can be allocated in either the host's or device's physical memory. Due to their conjoint address space both the host and device can access the buffer object using the same address.

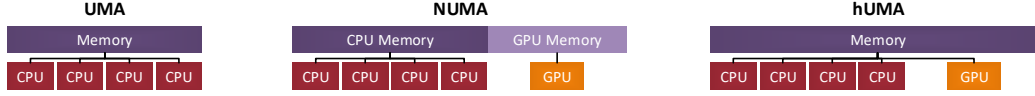


Fig. 1. UMA- and hUMA architectures with joint Address space. NUMA architecture with disjoint address spaces

III. DESIGN

In order to enable the CPU and GPU copy free memory access onto the payload buffer, it must be allocated using one of SocketCL's fine-grained allocation functions. These functions provide a comfortable wrapper interface to allocate fine-grained SVM buffers within SocketCL's context using OpenCL's `clSVMAlloc(...)` function.

A. Data Flow

Figure 3 is an approximation of the performed operations and data flow by abstracting implementation details. Additionally, it does visualize in which memory space (user-space or kernel-space) the memory is being located for each operation and which hardware component (CPU or GPU) performs them.

Upon initialization, SocketCL performs a preparative batch allocation represented in Figure 3 by the topmost block `allocate(batch)`. The memory for the batch and hence for the final packet fragments is located in the user-space. The allocation is performed by the CPU. In order to send data using SocketCL's send method a memory buffer has to be allocated using SocketCL's fine-grained allocation functions. This step is represented by the `allocate(payload)` block and executed by the CPU. The payload is also being allocated in the user-space. Each payload is eventually passed into SocketCL's send method represented by the `send` block. Within the send method the payload is being prepared for processing and information is being derived from its properties (e.g. size, number of fragments, etc.). Instead of processing each packet individually on the GPU, SocketCL gathers multiple packets in a batch represented by the `collect(payload)` block. If a minimum number of packets has been collected in a batch, the batch is then being scheduled for GPU processing. This step is represented by the `process(payload,batch)` block abstracting the individual protocols (UDP, IPv4, etc.). Each protocol is executed on the GPU but still operating on user-space memory due to the usage of SVM. Initially, the source

payload and destination batch are being allocated in different memory regions within the user-space. Therefore, the payload has to be copied into the associated batch memory region. This operation is represented by the `transfer(payload,batch)` block and is being executed on the GPU.

While the GPU is processing the batch and preparing ready-to-send packets, the CPU waits for the GPU to signalize finalization. It then sends all packets contained in the prepared batch, represented by the `send(packet,batch)` block. SocketCL therefore uses a system send function that transfers the batch's packets located in user-space to kernel-space (KS). This transition is represented by the connection between the `send(packet)` block and the `transferTo(packet,KS)` block, crossing the user-space and kernel-space. From this point on, the operating system takes control over the packet and sends it via a NIC, represented by the `send(packet)` block.

B. IPv4 Batch Structure

Each kernel has an IPv4 batch pointer as an argument in order to access packets and to communicate information. A batch stores information about how many packets are contained, as well as the input Ethernet and IPv4 headers. It also contains an array of IPv4 packets. A packet stores the total length of the UDP payload plus the UDP header size. It also stores its IPv4 packet ID, its number of fragments, the maximum length for each fragment and an instance of the pseudo header needed for the UDP checksum calculation. Most importantly a pointer to the UDP header and payload is being stored. Furthermore, each packet stores an array of fragment buffers. The IPv4 fragment structure is a wrapper around a simple byte array that allows us to address each fragment by its index.

C. UDP Kernel Implementation

Measurements showed that having less than $32 * 64 = 2048$ bytes of payload is not worth parallelizing and has therefore been set as the parallelization threshold.

According to RFC1071 [3] the Internet checksum is being calculated by summing up consecutive 16 bit words, accumulating the partial sums into a 32 bit field. Therefore, the pseudo header is being prepended to the UDP header and payload. Afterwards, the potential carries located in the most significant 16 bits are being added to the lowest significant 16 bits until the carry is 0. Finally, this accumulation is being negated bitwise with its lowest significant 16 bits representing the checksum. For calculating the checksum we wrote some helper functions used throughout the implementation.

To implement the checksum calculation, we store a pointer to the UDP header that is followed by the actual payload.

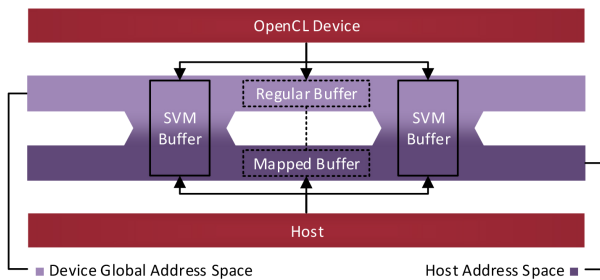


Fig. 2. Overlapping device and host address space

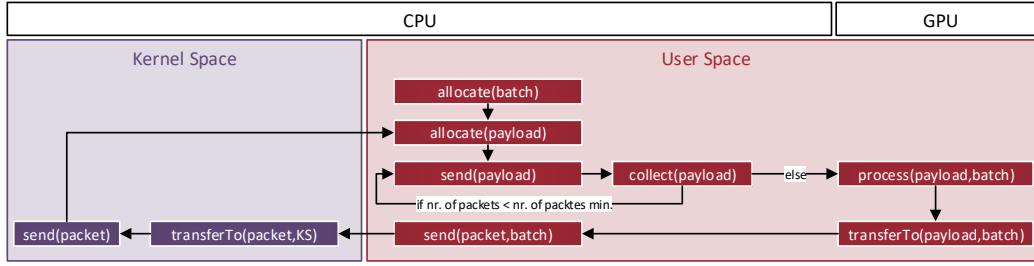


Fig. 3. SocketCL Data Flow Diagram

Afterwards, we calculate two partial sums for both the pseudo header and the UDP header and payload. Finally, we calculate the whole checksum by passing the two partial sums into the appropriate checksum function. The final checksum is then copied into the checksum field pointed by the packet's UDP header.

As in the serial version of the checksum calculation kernel, the pseudo header is being accumulated beforehand, but in this case into a global accumulation array of 32 bit fields. In case that the UDP payload size is not a multiple of 64, the remaining UDP payload size modulo 64 bytes are being accumulated before launching the child kernel. This way branch divergence is being reduced within the child kernel since each loop has the same number of iterations. In order to utilize all four SIMD units of a compute unit optimally, the UDP payload is split into 64 chunks of equal size. For each chunk a child kernel thread is being launched on the device accumulating 16 bit words into a local accumulator field. Afterwards, this local partial sum is being added atomically to the global accumulation field. To finalize the checksum from the globally accumulated value, one additional child kernel with one thread is being launched. In order to keep sequential correctness, the finalizing child kernel has to wait until the previous child kernel has finished. It then performs the carry addition and checksum negation similar to the serial version. The checksum is then copied into the checksum field pointed by the packet's UDP header.

Figure 4 illustrates the principle of the above mentioned child kernels operating on the UDP payload. At the top of the figure there is the batch consisting of multiple packets. The batch represents the input for all work-items, but each work-item is associated with one packet within the batch. This association is represented by the black line at the top of the figure, connecting the batch and the work-item. Each work-item is an instance of the UDP kernel and therefore visually represented by the red rectangle. The actual UDP payload is accessed via the packet and is assumed to be correctly predefined in respect to the UDP source port, destination port and length according to RFC0768 [12]. The UDP payload is represented by the purple rectangles at the top of the work-item highlighting the UDP header section at the front of the rectangle. Since the UDP checksum is being calculated

by including the UDP header and its payload, the whole memory region of the header and payload is therefore logically partitioned into multiple chunks. For each of these chunks a child kernel is launched to calculate the partial sum of that chunk. This is represented by the purple rectangle as the whole memory region transitioning into multiple purple rectangles by connecting them with black arrows. Here each black arrow represents a child kernel thread operating on a smaller purple rectangle as the memory chunk. At the end, all partial sums are being summed up and finalized into the actual UDP checksum. With the termination of the child kernel threads, the parent kernel does terminate as well [9]. The final UDP checksum is being stored in the associated UDP header entry, visualized by the orange arrow.

D. IPv4 Kernel Implementation

The IPv4 kernel does the following two tasks: defining the IPv4 header of each fragment and transferring the input payload into the fragment buffers. SocketCL defines an incomplete constant template IPv4 header that is being completed in the kernel by calculating each fragment's total length, byte offset and checksum. Afterwards, the IP fragmentation is done by copying the fragment's portion of the payload into the fragment's buffer. At this point, we can exploit the fact that by definition the IPv4 offset is an eight byte offset [13]. This means that we can copy chunks of 64 bits very fast.

At the beginning of the IPv4 kernel we decide whether to process this packet serially or in parallel. If the number of packet fragments is less than 8 we continue serially and parallel otherwise.

The parallel version of the IPv4 kernel launches a child kernel with one thread for each packet fragment. Each child kernel thread defines the IPv4 header for the fragment and copies the associated payload according to section III-D. Figure 5 shows the parallel IPv4 kernel and how it is fragmenting its payload into multiple chunks. At the top of the figure there is the batch consisting of multiple packets. Each work-item is associated with one packet from within the batch. The purple topmost rectangle represents the packet's csIPv4 payload that is being fragmented into multiple chunks. Each arrow represents a child kernel thread operating on a payload chunk and an IP individual header. The child kernel threads define the IPv4

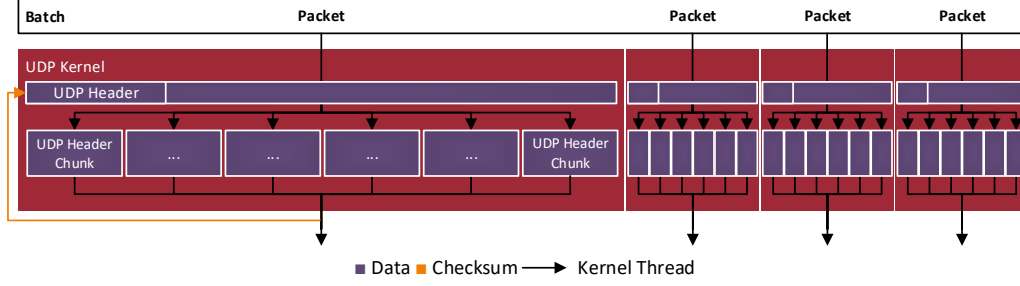


Fig. 4. Parallel UDP Kernel

header entries and afterwards calculate the IPv4 checksum. The headers are then copied into the appropriate fragment's location followed by the associated payload chunk. This process is visualized by the orange rectangle at the bottom of the threads, representing the destination fragment buffer for the individual child kernel thread.

E. Ethernet Kernel Implementation

The Ethernet kernel copies a constant Ethernet header to the base address of each packet fragment. To accelerate the copy process the Ethernet header is being copied in 64 bit, 32 bit and 16 bit chunks since it has $112 \text{ bit} = 14 \text{ bytes}$.

F. Buffer Reuse

In order to reduce buffer allocation time, the batch structure is allocated at the beginning of the program. The maximum number of fragments has been calculated to be 45, according to the following restrictions. With an MTU of 1500 byte and the IPv4 header having 20 bytes the maximum number of bytes in an IPv4 fragment is $1500_{MTU} - 20_{IPv4} = 1480 \text{ byte}$. Since IPv4 defines the byte offset to be an eight byte offset [13] we can copy chunks of 64 bit very fast. Another limitation is IPv4's 16 bit total length field [13]. The total length field defines the number of IPv4 header bytes plus the number of bytes in the payload. Therefore, the maximum number of fragments can be calculated by $\lceil (2^{16} - 1) / 1480 \rceil = 45$.

G. Dynamic Parallelism

Each kernel has only a batch pointer as argument and identifies its associated packet by its global ID within the work-group. The reason for that is to allow each packet thread to schedule child threads on the device depending on the number of packet fragments. The idea behind this is to utilize the GPUs hardware parallelism best by allowing the kernel to optimize the number of work-items and work-groups according to the problem size.

IV. EVALUATION

In this chapter we will present and discuss our measurement results. There are two major variables that are being configured throughout the measuring series: the batch size and the payload size. The batch size represents the minimum number of packets

that are being stored in a batch before it is processed on the GPU. The payload, however, defines the amount of data to be transferred. We will explain the results by relating them to our implementation's properties and our hardware's limitations.

A. Testbed

The APU used in our system is an AMD A10-7890K (CPU) with Radeon R7 (GPU) on the same die. The die holds four "Steamroller" CPU cores and eight GCN GPU cores. It is part of AMD's "Kaveri" series featuring HSA capabilities. One key capability is hUMA allowing the CPU and GPU to access shared memory by using a conjoint address space [2]. GCN, however, is a hardware architecture [1] that specifies properties of the GPU's compute units. Since our Radeon R7 is GCN compliant we know that each of our eight compute units has four SIMD units capable of executing one operation on 16 work-items at once.

Therefore, we can derive our system's specification as follows: one GPU has 8 compute units (cores), one compute unit has four SIMD units, each SIMD unit is 16-lanes wide and therefore executes the same operation on 16 work-items concurrently. The maximum number of possible work-item concurrency on our system can be calculated as follows: $8_{Compute Unit} * 4_{SIMD Unit} * 16_{Lane Width} = 512 \text{ work-items}$ executed concurrently or in other words: $1_{Compute Unit} * 4_{SIMD Unit} * 16_{Lane Width} = 64 \text{ work-items}$ per compute unit (core) execute concurrently. However, the actual distribution of work-items depends on the scheduler [1].

We tested our code from within Eclipse, using GDB as the debugger. After finalizing the project we built it using CMake and executed it from Ubuntu's command line terminal. In order to examine the correct number of fragments as well as valid IPv4 and UDP checksums, we used Wireshark 1.10.6. It allows us to display packets from a NIC's data stream according to filters and examine each protocol's data.

In order to measure the execution of CPU based portions, we used C++ 11's high resolution clock and time points. We identified crucial sections of the code and measured them isolated from the rest of the program. For the GPU based execution times we used OpenCL's profiling mechanisms. Enabling our command queue's profiling option allowed us to determine a kernel's start and end time point. Subtracting the

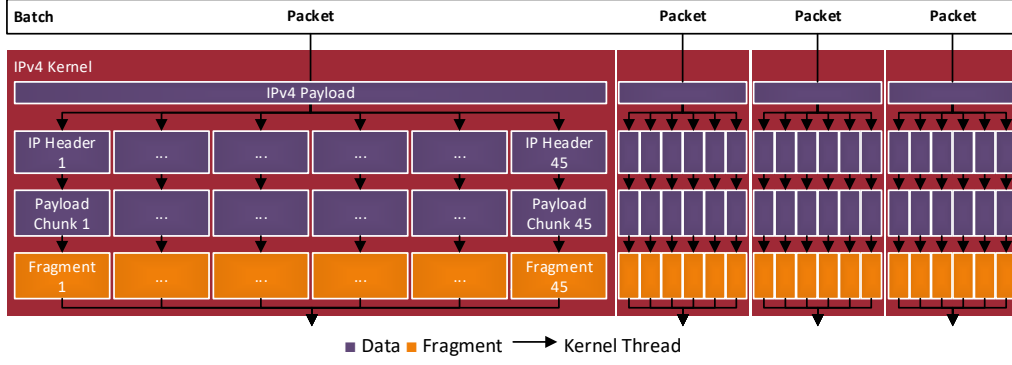


Fig. 5. Parallel IPv4 Kernel

start from the end time point results in the kernel's execution time in nanoseconds. For both the CPU's and GPU's execution time numerous iterations have been measured and averaged, using a 5% truncated mean. During all measurements the CPU was clocked at 4.1 GHz while the GPU's clock rate was controlled by the system itself.

B. Baseline

In order to determine speedup or slowdown of the implementation, we measured the system's socket send method time with increasing payload size and compared it to our raw socket's and SocketCL's execution times. The raw socket was implemented as a CPU based version of the Ethernet, IPv4 and UDP protocols to get an idea of each protocol's execution time.

SocketCL is built around a raw socket using its sendto method. Figure 6 shows proportional execution times of a raw socket's processing steps. To set these execution times in perspective, we compared them to a system UDP socket's sendto method. The raw socket implementation is straightforward and not optimized, except the usage of the kernel level checksum implementation in assembler. Figure 6 clearly shows the sendto section being by orders of magnitude slower than the Ethernet, IPv4 and UDP's checksum calculations. The diagram also shows that our raw socket implementation is in average as fast as the system implementation and therefore a realistic baseline. Although the sendto method outweighing Ethernet, IPv4 and UDP's checksum sections overall, its exponential increase at sending 4096 bytes is quite remarkable. Further investigation revealed that both the UDP socket and the raw socket use a send buffer of 212992 bytes.

In order to avoid a buffer overload, we query the number of bytes available in each socket's write buffer and wait for enough free memory before writing into it. Figure 7 shows the optimized execution times of the system UDP socket compared to our raw socket. Despite being optimized, the sendto section still takes the longest. We assume memory management between user and kernel memory to be the reason.

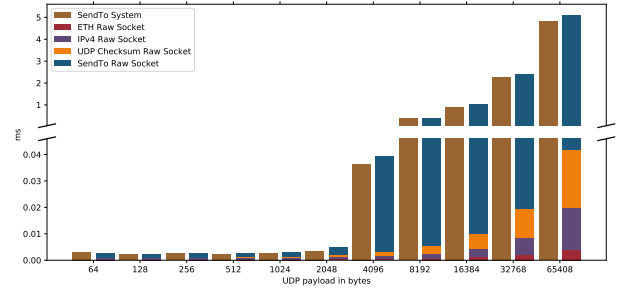


Fig. 6. System SendTo compared to Raw Socket execution time. Note split Y axis

However, we use the protocol processing times as baseline, since they are very close or faster than kernel level implementations, as shown by our measurements. We exclude the sendto parts from our further measurements, since it is identical for CPU and GPU processing.

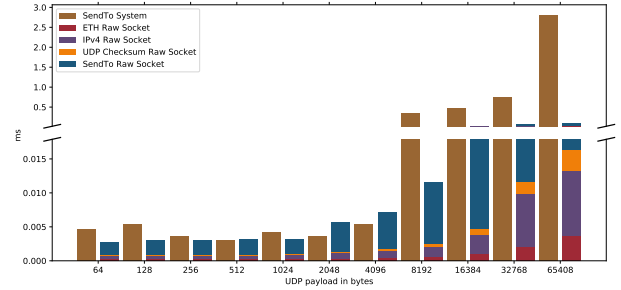


Fig. 7. UDP Socket and Raw Socket SendTo execution times

C. SocketCL Batch Execution

Figure 8 shows a comparison between the raw socket and SocketCL. The batch size in this comparison is one. Consequently, there was only one thread executing the protocol stack on the GPU. Despite increasing the payload size by the factor of two, the execution time of the Ethernet kernel

stays nearly the same. Correlating to the IPv4 kernel threshold, its execution time increases until 16384 bytes of payload. This is due to the fact that 16384 bytes of payload requires $\lceil (16384_{\text{Payload}} + 8_{\text{UDP}}) / (1500_{\text{MTU}} - 20_{\text{IPv4}}) \rceil = 12$ fragments, therefore satisfying the threshold condition. We can examine the UDP kernel's execution time analogously. With the UDP kernel threshold condition depending on the UDP payload size, we can see a slight increase in the UDP kernel execution time below 2048 bytes of payload. With 2048 bytes and more payload the UDP kernel threshold condition is satisfied according to $\forall n \in \mathbb{N} \lceil (2048 + n) / 64 \rceil \geq 32$. Therefore, the checksum calculation is done in parallel which is why there is no significant increase in execution time until 65408 bytes payload. It is important to keep in mind that the IPv4 kernel does not only define each fragment's IPv4 header but also transfers each fragment's payload. Unfortunately, in this configuration the raw socket's execution times are by orders of magnitude shorter than SocketCL's. These extreme results were expected for this configuration and provide a useful baseline for later comparisons.

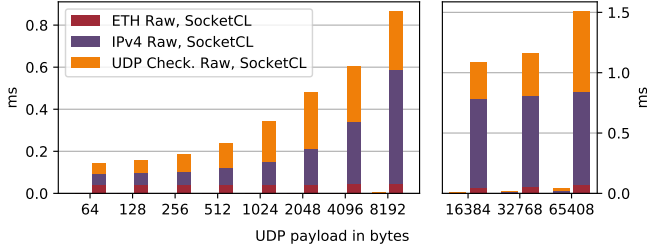


Fig. 8. Raw Socket (left) compared to SocketCL (right) batch execution time; Batch Size 1

The next configuration of interest is the one with batch size 64, shown in Figure 9). Since one compute unit has four 16-lane wide SIMD units the possible maximum number of concurrent thread operations on the compute unit is $4 \times 16 = 64$ [1]. Therefore, the execution times of SocketCL only increase slightly since there was still unused parallelization potential on the used compute unit. According to the OpenCL 2.0 specification "a work-group executes on a single compute unit" (cf. The OpenCL Specification [9]). Within SocketCL's send method we enqueue our protocol kernels as follows: number of global threads = current batch size, number of local threads = current batch size or 64 if current batch size is greater than 64. Therefore, we have one work-group since the work-group size equals the number of global threads. The execution times of the raw socket are also increasing slowly but are still more than twice as low as SocketCL's.

Increasing the batch size by 64 stepwise up to 256 caused the raw socket's total execution times to converge against SocketCL's for payload sizes up to 1024 bytes. Figure 10 shows similar total execution times around 0.25 ms with different distributions across the individual protocols. Above 1024 bytes payload SocketCL's execution times are still more than twice as slow, compared to the raw socket for certain

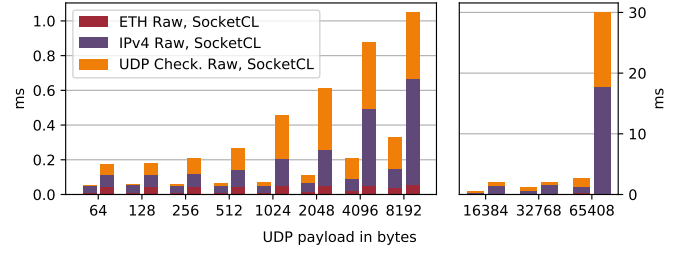


Fig. 9. Raw Socket (left) compared to SocketCL (right) batch execution time; Batch Size 64

payload sizes.

It is important to keep in mind that a batch size of 256 only utilizes four of our eight compute units since the 256 packets are being partitioned into $256/64 = 4$ work-groups running in parallel across four out of eight compute units. Therefore, a batch size of 256 only utilizes 50% of the potential hardware parallelism. Nevertheless, we already found a configuration where both implementations are nearly equally fast in producing ready-to-send packets.

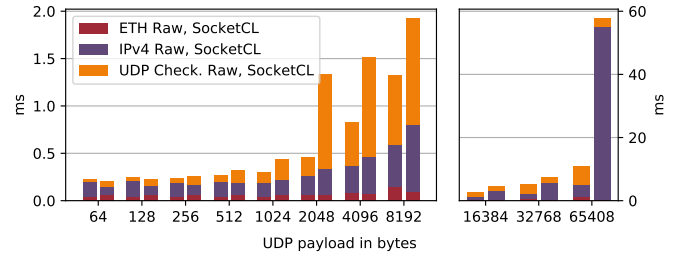


Fig. 10. Raw Socket (left) compared to SocketCL (right) batch execution time; Batch Size 256

In order to utilize all eight compute units we configured the batch size to 512 (see Figure 11) resulting in $512/64 = 8$ work-groups of 64 work-items. The raw socket's execution time only slightly increases but SocketCL's does decrease for all payload sizes up to 1024 bytes. At this point SocketCL is nearly twice as fast as the raw socket. The packet production rate in this configuration using 1024 bytes payload can be calculated as follows: $1s/0.5ms = 2000$ iterations per second hence $2000 \times 512_{\text{batch size}} \times 1024_{\text{byte}} \times 8_{\text{bit}} = 8.388608 \times 10^9 \text{ bit}$ therefore ca. 8.4 Gb/s.

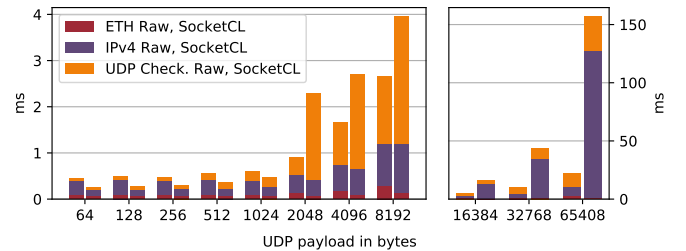


Fig. 11. Raw Socket (left) compared to SocketCL (right) batch execution time; Batch Size 512

Increasing the batch size to 1024 (see Figure 12) on average doubles SocketCL's execution time for all payload sizes. Optimistically rounding down the execution time of 512 bytes payload to 1 ms results in the following peak data rate: $1s/1ms = 1000$ iterations per second hence $1000 * 1024_{batch\ size} * 512_{byte} * 8_{bit} = 4.194304 * 10^9\ bit$ therefore ca. 4.2 Gb/s. This data rate is about the same as for the batch size of 512 and 512 bytes payload. However, the configuration in Figure 12 performs more than twice as bad compared to the configuration in Figure 11 for a payload size of 1024 bytes. Therefore, doubling the batch size from 512 to 1024 does not provide significant improvements for the protocol execution times. Although still outperforming the raw socket for payloads up to 256 bytes, the doubling of the batch size also doubles the overall execution times and is therefore not scaling well.

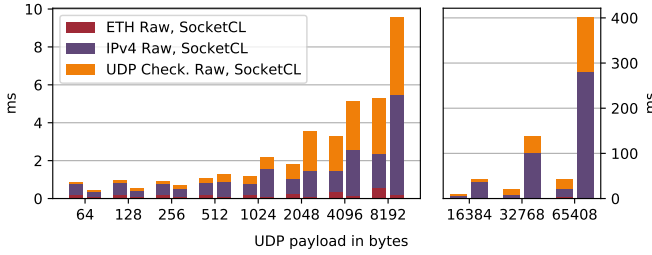


Fig. 12. Raw Socket compared to SocketCL batch execution time; Batch Size 1024

The sudden increase of the UDP and IPv4 kernel execution time in Figure 9 is remarkable. We assume that this is a result from cache repression. According to the GCN specification [1] each compute unit has an L1 cache of 16k bytes with 64 byte cache lines. The L2 cache, however, is sliced up into multiple 64-128k byte sized channels. With a payload size of about 64k bytes both, the L1 and L2 cache are being filled up very fast. Since our UDP and IPv4 kernel launch child kernels with up to 64 threads, there are $64 * 64_{bytes\ cache\ line} = 4096$ bytes of 16384 bytes already cached in L1, only after the first memory access of one child kernel thread. With e.g. a batch size of 64, there are $64_{packets} * 64_{threads} = 4096$ threads on one compute unit. These threads concurrently and consecutively access memory and therefore use the cache. Because of that, presume that cache repression, resulting from oversubscription, consequently leads to deteriorated performance in configurations with batch sizes of 64 up to 1024.

D. SVM Data Transfers

The values shown in Figure 13 have been measured on our system and are truncated means. The table lists execution times of data transfers done by the GPU and CPU. In case of the GPU we implemented a kernel only transferring data and measured its execution time using OpenCL's kernel profiling information. For the host execution time we implemented a primitive copy loop transferring eight bit chunks.

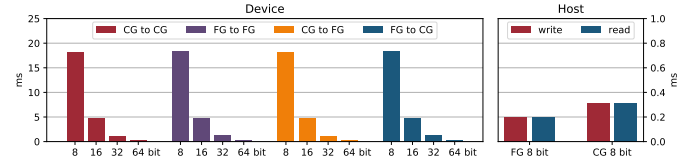


Fig. 13. Device and Host Fine- and Coarse-Grained memory transfer time

V. CONCLUSION AND FUTURE WORK

In this paper we presented SocketCL, a user-space network protocol stack acceleration. We leveraged the hUMA architecture together with OpenCL 2.0 to process network packets on a GPU. This frees up processing power on the CPU and speeds up the processing for high throughput scenarios.

The system configuration with a batch size of 512 performed best for our prototype-implementation. By outperforming the CPU raw socket implementation, it proves the feasibility to accelerate network packet generation on a GPU using OpenCL 2.0. The scalability can also be seen in the measurements: Until a batch size of 512 the total execution times for payloads up to 1024 bytes did not change significantly, although the number of packets was stepwise doubled.

Nevertheless, we still have some ideas how to improve our solution in future. As deployment solution we envision protocol virtualization solutions like VirtualStack [6], [8]. Therefore we plan to include SocketCL into VirtualStack.

Unfortunately, our evaluation environment does not support system SVM. Therefore, the operating system has to transfer our packets from the user-space to the kernel-space, although the packets are already prepared for sending. On a system capable of system SVM these packets could be initially allocated in the kernel-space or transferred there from the GPU. OpenCL 2.0 already provides an interface for it [9].

We decided to implement each protocol in an individual OpenCL kernel in order to be able to exchange protocols. If an application, uses a constant protocol stack a monolithic kernel implementation might lead to even better results.

Another restriction of our system is that we can not enqueue kernels for out-of-order execution. Since the protocols in our implementation are mostly independent from another, it would have been interesting to measure their execution times running in parallel on the device. OpenCL provides a mechanism for ensuring correct kernel execution orders by using event wait lists. If a kernel is enqueued with an event wait list, the kernel is not being executed unless all events from that list signal completion [9].

ACKNOWLEDGMENT

This work has been funded by the German Research Foundation (DFG) as part of the projects B2 within the Collaborative Research Center (CRC) 1053 – MAKI.

REFERENCES

- [1] White Paper — AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE, June 2012. https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.
- [2] D. Bouvier and B. Sander. Applying AMD's Kaveri APU for heterogeneous computing. In *2014 IEEE Hot Chips 26 Symposium (HCS)*, pages 1–42, 2014.
- [3] R. Braden, D. A. Borman, and C. Partridge. Computing the Internet checksum. *RFC*, 1071:1–24, 1988.
- [4] K. Fatahalian and M. Houston. A closer look at GPUs. *Commun. ACM*, 51(10):50–57, 2008.
- [5] S. Han, K. Jang, K. Park, and S. B. Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM*, pages 195–206, 2010.
- [6] J. Heuschkel, A. Frömmgen, J. Crowcroft, and M. Mühlhäuser. Virtualstack: Adaptive multipath support through protocol stack virtualization. In *11th International Networking Conference (INC)*, 2016, 2016.
- [7] J. Heuschkel, T. Hofmann, T. Hollstein, and J. Kuepper. Introduction to raw-sockets. 2017.
- [8] J. Heuschkel, L. Wang, E. Fleckstein, M. Ofenloch, M. Blöcher, J. Crowcroft, and M. Mühlhäuser. Virtualstack: Flexible cross-layer optimization via network protocol virtualization. In *43rd Local Computer Networks (LCN)*, 2015 IEEE, 2018.
- [9] L. Howes and A. Munshi. The OpenCL specification, version 2.0. *Khronos Group*, 2015. <https://www.khronos.org/registry/OpenCL/specs/opencvl-2.0.pdf>.
- [10] D. P. D. K. D. Intel. Programmers guide, 2014.
- [11] S. Mukherjee, Y. Sun, P. Blinzer, A. K. Ziabari, and D. R. Kaeli. A comprehensive performance analysis of HSA and OpenCL 2.0. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, pages 183–193, 2016.
- [12] J. Postel. User datagram protocol. STD 6, RFC Editor, August 1980.
- [13] J. Postel. Internet protocol. STD 5, RFC Editor, September 1981.
- [14] L. Rizzo. Netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.
- [15] P. Rogers and C. Fellow. AMD heterogeneous uniform memory access. *AMD Whitepaper*, 2013.
- [16] M. Tsou. Big data: techniques and technologies in geoinformatics. *Annals of GIS*, 20(4):295–296, 2014.