Raising the Bar for Using GPUs in Software Packet Processing

Anuj Kalia, Dong Zhou, Michael Kaminsky*, and David G. Andersen
*Carnegie Mellon University and *Intel Labs*

Abstract

Numerous recent research efforts have explored the use of Graphics Processing Units (GPUs) as accelerators for software-based routing and packet handling applications, typically demonstrating throughput several times higher than using legacy code on the CPU alone.

In this paper, we explore a new hypothesis about such designs: For many such applications, the benefits arise less from the GPU hardware itself as from the expression of the problem in a language such as CUDA or OpenCL that facilitates memory latency hiding and vectorization through massive concurrency. We demonstrate that in several cases, after applying a similar style of optimization to algorithm implementations, a CPU-only implementation is, in fact, *more resource efficient* than the version running on the GPU. To "raise the bar" for future uses of GPUs in packet processing applications, we present and evaluate a preliminary language/compiler-based framework called G-Opt that can accelerate CPU-based packet handling programs by automatically hiding memory access latency.

1 Introduction

The question of matching hardware architectures to networking requirements involves numerous trade-offs between flexibility, the use of off-the-shelf components, and speed and efficiency. ASIC implementations are fast, but relatively inflexible once designed, and must be produced in large quantities to offset the high development costs. Software routers are as flexible as code, but have comparatively poor performance, in packets-per-second (pps), as well as in cost (pps/$) and energy efficiency (pps/watt). Both ends of the spectrum are successful: Software-based firewalls are a popular use of the flexibility and affordability of systems up to a few gigabits per second; commodity Ethernet switches based on high-volume ASICs achieve seemingly unbeatable energy and cost efficiency.

In the last decade, several potential middle grounds emerged, from network forwarding engines such as the Intel IXP, to FPGA designs [12], and, as we focus on in this paper, to the use of commodity GPUs. Understanding the advantages of these architectures, and how to best exploit them, is important both in research (software-based implementations are far easier to experiment with) and in practice (software-based approaches are used for low-speed applications and in cases such as forwarding within virtual switches [13]).

Our goal in this paper is to advance understanding of the advantages of GPU-assisted packet processors compared to CPU-only designs. In particular, noting that several recent efforts have claimed that GPU-based designs can be faster even for simple applications such as IPv4 forwarding [23, 43, 31, 50, 35, 30], we attempt to identify the *reasons* for that speedup. At the outset of this work, we hypothesized that much of the advantage came from the way the GPUs were *programmed*, and that less of it came from the fundamental hardware advantages of GPUs (computational efficiency from having many processing units and huge memory bandwidth).

In this paper, we show that this hypothesis appears correct. Although GPU-based approaches are faster than a straightforward implementation of various forwarding algorithms, it is possible to transform the CPU implementations into a form that is more resource efficient than GPUs.

For many packet processing applications, the key advantage of a GPU is *not* its computational power, but that it can transparently hide the 60-200ns of latency required to retrieve data from main memory. GPUs do this by exploiting massive parallelism and using fast hardware thread switching to switch between sets of packets when one set is waiting for memory. We demonstrate that insights from code optimization techniques such as group prefetching and software pipelining [17, 51] apply to typical CPU packet handling code to boost its performance. In many cases, the CPU version is more resource efficient than the GPU, and delivers lower latency because it does not incur the additional overhead of transferring data to and from the GPU.

Finally, to make these optimizations more widely usable, both in support of practical implementations of software packet processing applications, and to give future research a stronger CPU baseline for comparison, we present a method to automatically transform data structure lookup code to overlap its memory accesses and computation. This automatically transformed code is up to 1.5-6.6x faster than the baseline code for several common

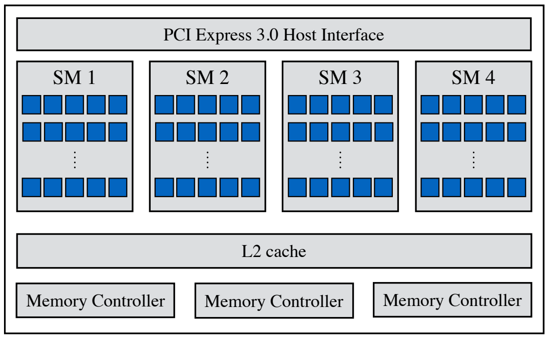


Figure 1: Simplified architecture of an NVIDIA GTX 650. The global memory is not shown.

lookup patterns, and its performance is within 10% of our hand-optimized version. By applying these optimizations, we hope to "raise the bar" for future architectural comparisons against the baseline CPU-based design.

2 Strengths and weaknesses of GPUs for packet processing

In this section, we first provide relevant background on GPU architecture and programming, and discuss the reasons why previous research efforts have used GPUs as accelerators for packet processing applications. Then, we show how the fundamental differences between the requirements of packet processing applications and conventional graphics applications make GPUs less attractive for packet processing than people often assume. Throughout this paper, we use NVIDIA and CUDA's terminology for GPU architecture and programming model, but we believe that our discussion and conclusions apply equally to other discrete GPUs (e.g., GPUs using OpenCL).

2.1 GPU strengths: vectorization and memory latency hiding

A modern CUDA-enabled GPU (Figure 1) consists of a large number of processing cores grouped into Streaming Multiprocessors (SMs). It also contains registers, a small amount of memory in a cache hierarchy, and a large global memory. The code that runs on a GPU is called a *kernel*, and is executed in groups of 32 threads called *warps*. The threads in a warp follow a SIMT (Single Instruction, Multiple Thread) model of computation: they share an instruction pointer and execute the same instructions. If the threads "diverge" (i.e., take different execution paths), the GPU selectively disables the threads as necessary to allow them to execute correctly.

Vectorization: The large number of processing cores on a GPU make it attractive as a vector processor for packets. Although network packets do have some inter-packet ordering requirements, most core networking functions such as lookups, hash computation, or encryption can be executed in parallel for multiple packets at a time. This parallelism is easily accessible to the programmer through well-established GPU-programming frameworks such as CUDA and OpenCL. The programmer writes code for a single thread; the framework automatically runs this code with multiple threads on multiple processors.

Comparison with CPUs: The AVX2 vector instruction set in the current generation of Intel processors has 256-bit registers that can process 8 32-bit integers in parallel. However, the programming language support for CPU-based vectorization is still maturing [8].

Memory latency hiding: Packet processing applications often involve lookups into large data structures kept in DRAM. Absent latency-hiding, access to these structures will stall execution while it completes (300-400 cycles for NVIDIA's GPUs). Modern GPUs hide latency using hardware. The warp scheduler in an SM holds up to 64 warps to run on its cores. When threads in a warp access global memory, the scheduler switches to a different warp. Each SM has *thousands* of registers to store the warp-execution context so that this switching does not require explicitly saving and restoring registers.

Comparison with CPUs: Three architectural features in modern CPUs enable memory latency hiding. First, CPUs have a small number of hardware threads (typically two) that can run on a single core, enabling ongoing computation when one thread is stalled on memory. Unfortunately, while each core can maintain up to ten outstanding cache misses [51], hyperthreading can only provide two "for free". Second, CPUs provide both hardware and software-managed prefetching to fetch data from DRAM into caches before it is needed. And third, after issuing a DRAM access, CPUs can continue executing independent instructions using out-of-order execution. These features, however, are less able to hide latency in unmodified code than the hardware-supported context switches on GPUs, and leave ample room for improvement using latency-hiding code optimizations (Section 3).

2.2 GPU weaknesses: setup overhead and random memory accesses

Although GPUs have attractive features for accelerating packet processing, two requirements of packet processing applications make GPUs a less attractive choice:

Many networking applications require low latency. For example, it is undesirable for a software router in a datacenter to add more than a few microseconds of latency [20]. In the measurement setup we use in this paper, the RTT through an unloaded CPU-based forwarder is 16µs. Recent work in high-performance packet processing reports numbers from 12 to 40µs [32, 51].

Unfortunately, merely communicating from the CPU to the GPU and back may add more latency than the total RTT of these existing systems. For example, it takes $\sim$ 15µs to transfer one byte to and from a GPU, and ~ 5µs to launch the kernel [33]. Moreover, GPU-accelerated systems must assemble large batches of packets to process on the GPU in order to take advantage of their massive parallelism and amortize setup and transfer costs. This batching further increases latency.

Networking applications often require random memory accesses into data structures, but the memory subsystem in GPUs is optimized for contiguous access. Under random accesses, GPUs lose a significant fraction of their memory bandwidth advantage over CPUs.

We now discuss these two factors in more detail. Then, keeping these two fundamental factors in mind, we perform simple experiments through which we seek to answer the following question: *When is it beneficial to offload random memory accesses or computation to a GPU?*

2.3 Experimental Setup

We perform our measurements on three CPUs and three GPUs, representing the low, mid, and high end of the recent CPU and GPU markets. Table 1 shows their relevant hardware specifications and cost. All prices are from `http://www.newegg.com` as of 9/2014. The K20 connects to an AMD Opteron 6272 socket via PCIe 2.0 x16, the GTX 980 to a Xeon E5-2680 via PCIe 2.0 x16, and the GTX 650 to an i7-4770 via PCIe 3.0 x16.

2.4 Latency of CPU-GPU communication

We first measure the minimum time required to involve a GPU in a computation—the minimum extra latency that a GPU in a software router will add to every packet. In this experiment, the host transfers an input array with N 32-bit integers to the GPU, the GPU performs negligible computations on the array, and generates an output array with the same size. To provide a fair basis for comparison with CPUs, we explored the space of possible methods for this CPU-GPU data exchange in search of the best, and present results from two methods here:

Asynchronous CUDA functions: This method performs memory copies and kernel launch using asynchronous functions (e.g., `cudaMemcpyAsync`) provided by the CUDA API. Unlike synchronous CUDA functions, these functions can reduce the total processing time by overlapping data-copying with kernel execution. Figure 2 shows the timing breakdown for the different functions. We define the time taken for an asynchronous CUDA function call as the time it takes to return control to the calling CPU thread. The extra time taken to complete all the pending asynchronous functions is shown separately.

Polling on mapped memory: To avoid the overhead of CUDA functions, we tried using CUDA's mapped memory feature that allows the GPU to access the host's memory over PCIe. We perform CPU-GPU communication using mapped memory as follows. The CPU creates the

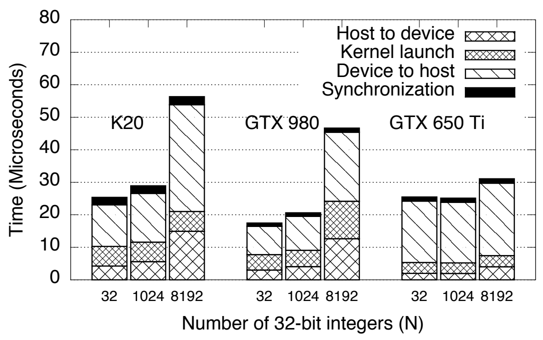


Figure 2: Timing breakdown of CPU-GPU communication with asynchronous CUDA functions.

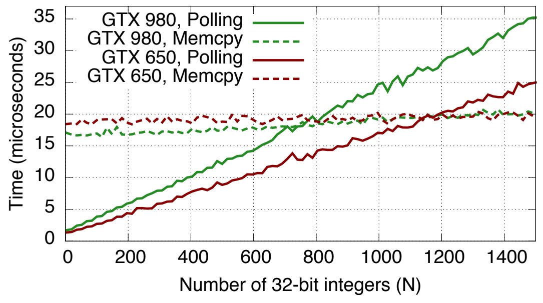


Figure 3: Minimum time for GPU-involvement with kernel-polling on mapped memory.

input array and a flag in the host memory and raises the flag when the input is ready. CUDA threads continuously poll the flag and read the input array when they notice a raised flag. After processing, they update the output array and start polling for the flag to be raised again. This method does not use any CUDA functions in the critical path, but all accesses to mapped memory (reading the flag, reading the input array, and writing to the output array) that come from CUDA threads lead to PCIe transactions.

Figure 3 shows the time taken for this process with different values of N. The solid lines show the results with polling on mapped memory, and the dotted lines use the asynchronous CUDA functions. For small values of N, avoiding the CUDA driver overhead significantly reduces total time. However, polling generates a linearly increasing number of PCIe transactions as N increases, and becomes slower than CUDA functions for $N \sim 1000$. As GPU-offloading generally requires larger batch sizes to be efficient, we only use asynchronous CUDA functions in the rest of this work.

2.5 GPU random memory access speed

Although GPUs have much higher *sequential* memory bandwidth than CPUs (Table 1), they lose a significant fraction of their advantage when memory accesses are random, as in data structure lookups in many packet processing applications. We quantify this loss by measuring

Name	# of cores	Memory b/w	Arch., Lithography	Released	Cost	Random Access Rate
Xeon E5-2680	8	51.2 GB/s	SandyBridge, 32nm	2012	$1,748	595 M/s
Xeon E5-2650 v2	8	59.7 GB/s	IvyBridge, 22nm	2013	$1,169	464 M/s
i7-4770	4	25.6 GB/s	Haswell, 22nm	2013	$309	262 M/s
Tesla K20	2,496	208 GB/s	Kepler, 28nm	2012	$2,848	792 M/s
GTX 980	2048	224 GB/s	Maxwell, 28nm	2014	$560	1260 M/s
GTX 650 Ti	768	86.4 GB/s	Kepler, 28nm	2012	$130	597 M/s

Table 1: CPU and GPU specifications, and *measured* random access rate

the random access rate of CPUs and GPU as follows. We create a 1 GB array `L` containing a random permutation of $\{0, \ldots, 2^{28} - 1\}$, and an array `H` of B random offsets into `L`, and pre-copy them to the GPU's memory. In the experiment, each element of `H` is used to follow a chain of random locations in `L` by executing `H[i] = L[H[i]]` D times. For maximum memory parallelism, each GPU thread handles one chain, whereas each CPU core handles all the chains simultaneously. Then, the random access rate is $\frac{B*D}{t}$, where t is the time taken to complete the above process.

Table 1 shows the rate achieved for different CPUs and GPUs with $D = 10$, and the value of B that gave the maximum rate ($B = 16$ for CPUs, $B = 2^{19}$ for GPUs).[1] Although the advertised memory bandwidth of a GTX 980 (224 GB/s) is 4.37x of a Xeon E5-2680, our measured random access rate is only 2.12x. This reduction in GPU bandwidth is explained by the inability of its memory controller to coalesce memory accesses done by different threads in a warp. The coalescing optimization is only done when the warp's threads access contiguous memory, which rarely happens in our experiment.

2.6 When should we offload to a GPU?

Given that involving GPUs takes several microseconds, and their random memory access rate is not much higher than that of CPUs, it is intriguing to find out in which scenarios GPU-offloading is really beneficial. Here, we focus on two widely-explored tasks from prior work: random memory accesses and expensive computations. In the rest of this paper, all experiments are done on the E5-2680 machine with the GTX 980 GPU.

2.6.1 Offloading random memory accesses

Lookups in pointer-based data structures such as IPv4/IPv6 tries and state machines follow a chain of mostly random pointers in memory. To understand the benefit of offloading these memory accesses to GPUs, we perform the experiment in Section 2.5, *but include the time taken to transfer* `H` *to and from the GPU*. `H` represents a batch of header addresses used for lookups in packet processing. We set B (the size of the batch) to 8192—slightly higher than the number of packets arriving in 100µs on our 40 Gbps network. We use different values of D, representing the variation in the number of pointer-dereferencing operations for different data structures.

Figure 4a plots the number of headers processed per second for the GPU and different numbers of CPU cores. As D increases, the overhead of the CUDA function calls gets amortized and the GPU outperforms an increasing number of CPU cores. However for $D \leq 4$, the CPU outperforms the GPU, indicating that offloading ≤ 4 dependent memory accesses (e.g., IPv4 lookups in PacketShader [23] and GALE [50]) should be slower than using the CPU only.

2.6.2 Offloading expensive computation

Although GPUs can provide substantially more computing power than CPUs, the gap decreases significantly when we take the communication overhead into account. To compare the computational power of GPUs and CPUs for varying amounts of offloaded computation, we perform a sequence of D dependent `CityHash32` [4] operations on a each element of `H` (B is set to 8192).

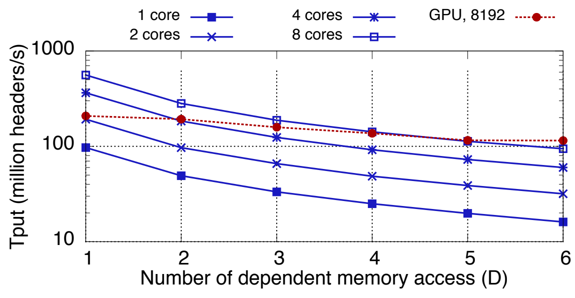
Figure 4b shows that the CPU outperforms the GPU if $D \leq 3$. Computing 3 CityHashes takes $\sim$ 40ns on one CPU core. This time frame allows for a reasonable amount of computation before it makes sense to switch to GPU offloading. For example, a CPU core can compute the cryptographically stronger Siphash [16] of a 16-byte string in $\sim$ 36ns.

3 Automatic DRAM latency hiding for CPUs

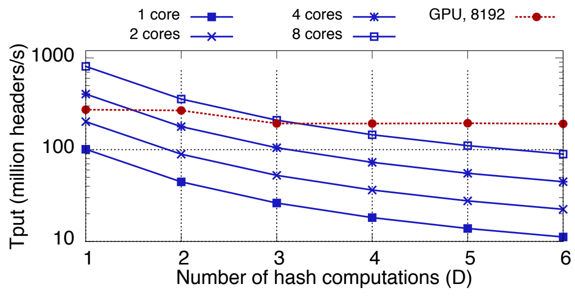
The section above showed that CPUs support respectable random memory access rates. However, *achieving* these rates is challenging: CPUs do not have hardware support for fast thread switching that enables latency hiding on GPUs. Furthermore, programs written for GPUs in CUDA or OpenCL start from the perspective of processing many (mostly)-independent packets, which facilitates latency hiding.

The simple experiment in the previous section saturated the CPU's random memory access capability because of its simplicity. Our code was structured such that each core issued B independent memory accesses—one for each chain—in a tight loop. The CPU has a limited window for reordering and issuing out-of-order instructions.

[1]The K20's rate increases to 1390 M/s if `L` is smaller than 256 MB.



***(a)** Offloading random memory accesses*



***(b)** Offloading expensive computation*

Figure 4: Comparison of CPU and GPU performance for commonly offloaded tasks. Note the log scale on Y axis.

When memory accesses are independent and close in the instruction stream, the CPU *can* hide the latency by issuing subsequent accesses before the first completes. However, as described below, re-structuring and optimizing real-world applications in this manner is tedious or inefficient.

A typical unoptimized packet-processing program operates by getting a batch of packets from the NIC driver, and then processing the packets one by one. Memory accesses *within* a packet are logically dependent on each other, and the memory accesses across multiple packets are spaced far apart in the instruction stream. This reduces or eliminates the memory latency-hiding effect of out-of-order execution. Our goal, then, is to (automatically) restructure this CPU code in a way that hides memory latency.

In this section, we first discuss existing techniques for optimizing CPU programs to hide their memory access latency. As these techniques are not suited to *automatically* hiding *DRAM* latency, we present a new technique called G-Opt that achieves this goal for programs with parallel data structure lookups. Although the problem of automatic parallelization and latency hiding in general is hard, certain common patterns in packet processing applications *can* be handled automatically. G-Opt hides the DRAM latency for parallel lookups that observe the same constraints as their CUDA implementations: independence across lookups and read-only data structures.

```
 1 find(entry_t *h_table, key_t *K,value_t *V) {
 2     int i;
 3     for(i = 0; i < B; i ++) {
 4         int entry_idx = hash(K[i]);
 5         // g_expensive (&h_table[entry_idx]);
 6         value_t *v_ptr = h_table[entry_idx].v_ptr;
 7         if(v_ptr != NULL) {
 8             // g_expensive (v_ptr);
 9             V[i] = *v_ptr;
10         } else {
11             V[i] = NOT_FOUND;
12         }
13     }
14 }
```

Figure 5: Naive batched hash table lookup.

3.1 Existing techniques for hiding memory access latency

3.1.1 Group prefetching

Group prefetching hides latency by processing a batch of lookups at once and by using *memory prefetches* instead of memory accesses. In a prefetch, the CPU issues a request to load a given memory location into cache, but does not wait for the request to complete. By intelligently scheduling independent instructions after a prefetch, useful work can be done while the prefetch completes. This “hiding” of prefetch latency behind independent instructions can increase performance significantly.

A data structure lookup often consists of a series of dependent memory accesses. Figure 5 shows a simple implementation of a batched hash table lookup function. Each invocation of the function processes a batch of **B** lookups. Each hash table entry contains an integer key and a pointer to a value. For simplicity, we assume for now that there are no hash collisions. There are three steps for each lookup in the batch: hash computation (line 4), accessing the hash table entry to get the value pointer (line 6), and finally accessing the value (line 9). Within a lookup, each step depends on the previous one: there are no independent instructions that can be overlapped with prefetches. However, independent instructions do exist if we consider the different lookups in the batch [17, 51].

Figure 6 is a variant of Figure 5 with the *group prefetching* optimization. It splits up the lookup code into three stages, delimited by the expensive memory accesses in the original code. We define an expensive memory access as a memory load operation that is likely to miss all levels of cache and hit DRAM. The optimized code does not directly access the hash table entry after computing the hash for a lookup key; it issues a prefetch for the entry and proceeds to compute the hash for the remaining lookups. By doing this, it does not stall on a memory lookup for the hash table entry and instead overlaps the prefetch with independent instructions (hash computation and prefetch instructions) from other lookups.

```c
find(entry_t *h_table, key_t *K,value_t *V) {
    int entry_idx[B], i;
    value_t *v_ptr[B];
    // Stage 1: Hash-Computation
    for(i = 0; i < B; i ++) {
        entry_idx[i] = hash(K[i]);
        prefetch(&h_table[entry_idx[i]]);
    }

    // Stage 2: Access hash table entry
    for(i = 0; i < B; i ++) {
        v_ptr[i] = h_table[entry_idx[i]].v_ptr;
        prefetch(v_ptr[i]);
    }

    // Stage 3: Access value
    for(i = 0; i < B; i ++) {
        if(v_ptr[i] != NULL) {
            V[i] = *v_ptr[i];
        } else {
            V[i] = NOT_FOUND;
        }
    }
}
```

Figure 6: Batched lookup with group prefetching.

Unfortunately, group prefetching does not apply trivially to general lookup code because of control divergence. It requires dividing the code linearly into stages, which is difficult for code with complicated control flow. Even if such a linear layout were possible, control divergence will require a possibly large number of *masks* to record the execution paths taken by different lookups. Divergence also means that fewer lookups from a batch will enter later stages, reducing the number of instructions available to overlap with prefetches.

3.1.2 Fast context switching

In Grappa [36], fast context switching among lightweight threads is used to hide the latency of remote memory accesses over InfiniBand. After issuing a remote memory operation, the current thread yields control in an attempt to overlap the remote operation's execution with work from other threads. The minimum reported context switch time, 38 nanoseconds, is sufficiently small compared to remote memory accesses that take a few microseconds to complete. Importantly, this solution (like the hardware context switches on GPUs) is able to handle the control divergence of general packet processing. Unfortunately, the local DRAM accesses required in most packet processing applications take 60-100 nanoseconds, making the overhead of even highly optimized generic context switching unacceptable.

3.2 G-Opt

We now describe our method, called *G-Opt*, for automatically hiding DRAM latency for data structure lookup algorithms. Our technique borrows from both group prefetching and fast context switching. Individually, each of these techniques falls short of our goal: Group prefetching can hide DRAM latency but there is no general technique to automate it, and fast context switching is easy to automate but has large overhead.

G-Opt is a source-to-source transformation that operates on a batched lookup function, $\mathcal{F}$, written in C. It imposes the same constraints on the programmer that languages such as CUDA [3], OpenCL, and Intel's ISPC [8] do: the programmer must write *batch* code that expresses parallelism by granting the language explicit permission to run the code on multiple independent inputs. G-Opt additionally requires the programmer to annotate the expensive memory accesses that occur within $\mathcal{F}$. To annotate the batch lookup code in Figure 5, the lines with `g_expensive` hints should be uncommented, indicating that the following lines (line 6 and line 9) contain an expensive memory access. `g_expensive` is a macro that evaluates to an empty string: it does not affect the original code, but G-Opt uses it as a directive during code generation. The input function, $\mathcal{F}$ processes the batch of lookups one-by-one as in Figure 5. Applying G-Opt to $\mathcal{F}$ yields a new function $\mathcal{G}$ that has the same result as $\mathcal{F}$, but includes extra logic that tries to hide the latency of DRAM accesses. Before describing the transformation in more detail, we outline how the function $\mathcal{G}$ performs the lookups.

$\mathcal{G}$ begins by executing code for the first lookup. Instead of performing an expensive memory access for this lookup, $\mathcal{G}$ issues a prefetch for the access and switches to executing code for the second lookup. This continues until the second lookup encounters an expensive memory access, at which point $\mathcal{G}$ switches to the third lookup, or back to the first lookup if there are only two lookups in the batch. Upon returning to the first lookup, the new code then accesses the memory that it had previously prefetched. In the optimal case, this memory access does not need to wait on DRAM because the data is already available in the processor's L1 cache.

We now describe the transformation in more detail by discussing its action on the batched hash table lookup code in Figure 5. The code produced by G-Opt is shown in Figure 7. The key characteristics of the transformed code are:

1. **Cheap per-lookup state-maintenance**: There are two pieces of state for a lookup in $\mathcal{G}$. First, the function-specific state for a lookup is maintained in local arrays derived from the local variables in $\mathcal{F}$: the local variable named `x` in $\mathcal{F}$ is stored in `x[I]` for the `I`th lookup in $\mathcal{G}$. Second, there are two G-Opt-specific control variables for lookup `I`: `g_labels[I]` stores its `goto` target, and `g_mask`'s `I`th bit records if it has finished execution.

2. **Lookup-switching using gotos**: Instead of stalling on a memory access for lookup `I`, $\mathcal{G}$ issues a prefetch for the memory address, saves the `goto` target at the next line of code into `g_labels[I]`, and jumps to the `goto` target for the next lookup. We call this procedure a "Prefetch, Save, and Switch", or PSS. It acts as a fast switching mechanism between different lookups, and is carried out using the `G_PSS` macro that takes two arguments: the address to prefetch and the label to save as the `goto` target. G-Opt inserts a `G_PSS` macro and a `goto` target before every expensive memory access; this is achieved by using the annotations in $\mathcal{F}$.
3. **Extra initialization and termination code**: G-Opt automatically sets the initial `goto` target label for all lookups to `g_label_0`. Because different lookups can take significantly different code paths in complex applications, they can reach the label `g_end` in any order. $\mathcal{G}$ uses a bitmask to record which lookups have finished executing, and the function returns only after all lookups in the batch have reached `g_end`.

We implemented G-Opt using the ANTLR parser generator [2] framework. G-Opt performs 8 passes over the input function's Abstract Syntax Tree. It converts local variables into local arrays. It recognizes the annotations in the input function and emits labels and `G_PSS` macros. Finally, it deletes the top-level loop (written as a `foreach` loop to distinguish it from other loops) and adds the initialization and termination code based on the control variables. Our current implementation does not allow pre-processor macros in the input code, and enforces a slightly restricted subset of the ISO C grammar to avoid ambiguous cases that would normally be resolved subsequent to parsing (e.g., the original grammar can interpret `foo(x);` as a variable declaration of type `foo`).

```c
// Prefetch, Save label, and Switch lookup
#define G_PSS(addr, label) do {
    prefetch(addr); \
    g_labels[I] = &&label; \
    I = (I + 1) % B;  \
    goto *g_labels[I]; \
} while(0);

find(entry_t *h_table, key_t *K,value_t *V) {
    // Local variables from the function
    int entry_idx[B];
    value_t *v_ptr[B];

    // G-Opt control variables
    int I = 0, g_mask = 0;
    void *g_labels[B] = {g_label_0};

g_label_0:
    entry_idx[I] = hash(K[I]);
    G_PSS(&h_table[entry_idx[I], g_label_1);
g_label_1:
    v_ptr[I] = h_table[entry_idx[I]].v_ptr;
    if(v_ptr[I] != NULL) {
        G_PSS(v_ptr[I], g_label_2);
g_label_2:
        V[I] = *v_ptr[I];
    } else {
        V[I] = NOT_FOUND;
    }

g_end:
    g_labels[I] = &&g_end;
    g_mask = SET_BIT(g_mask, I);
    if(g_mask == (1 << B) - 1) {
        return;
    }
    I = (I + 1) % B;
    goto *g_labels[I];
}
```

Figure 7: Batched hash table lookup after G-Opt transforming.

3.3 Evaluation of G-Opt

In this section, we evaluate G-Opt on a collection of synthetic microbenchmarks that perform random memory accesses; Section 4 discusses the usefulness of G-Opt for a full-fledged software router. We present a list of our microbenchmarks along with their possible uses in real-world applications below. For each microbenchmark, we also list the source of expensive memory accesses and computation. The speedup provided by G-Opt depends on a balance between these two factors: G-Opt is not useful for compute-intensive programs with no expensive memory accesses, and loses some of its benefit for memory-intensive programs with little computation.

Cuckoo hashing: Cuckoo hashing [37] is an efficient method for storing in-memory lookup tables [19, 51]. Our 2-8 cuckoo hash table (using the terminology from MemC3 [19]) maps integer keys to integer values. *Computation*: hashing a lookup key. *Memory*: reading the corresponding entries from the hash table.

Pointer chasing: Several algorithms that operate on pointer-based data structures, such as trees, tries, and linked lists, are based on following pointers in memory and involve little computation. We simulate a pointer-based data structure with minimal computation by using the experiment in Section 2.5. We set D to 100, emulating the long chains of dependent memory accesses performed for traversing data structures such as state machines and trees. *Computation*: negligible. *Memory*: reading an integer at a random offset in `L`.

IPv6 lookup: To demonstrate the applicability of G-Opt to real-world code, we used it to accelerate Intel DPDK's batched IPv6 lookup function. Applying G-Opt to the lookup code required only minor syntactic changes and one line of annotation, whereas hand-optimization required significant changes to the code's logic. We populated DPDK's Longest Prefix Match (LPM) structure with 200,000 random IPv6 prefixes (as done in Packet-

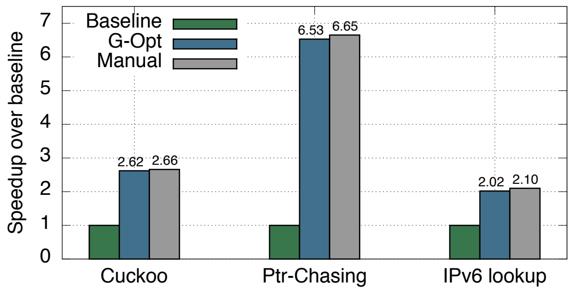


Figure 8: Speedup with G-Opt and manual group prefetching.

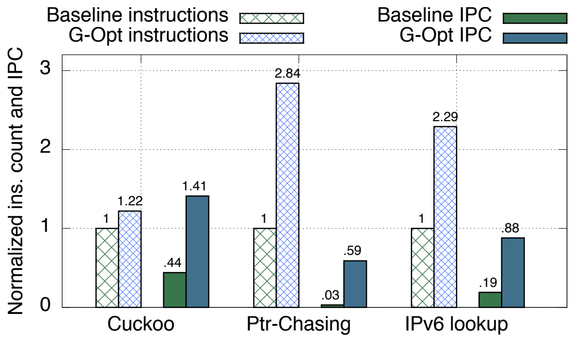


Figure 9: Instruction count and IPC with G-Opt.

Shader [23]) with lengths between 48 and 64 bits,[2] and used random samples from these prefixes to simulate a worst case lookup workload. *Computation*: a few arithmetic and bitwise operations. *Memory*: 4 to 6 accesses to the LPM data structure.

Our microbenchmarks use 2 MB hugepages to reduce TLB misses [32]. We use `gcc` version 4.6.3 with -O3. The experiments in this section were performed on a Xeon E5-2680 CPU with 32 GB of RAM and 20 MB of L3 cache. We also tested G-Opt on the CPUs in Table 1 with similar results.

3.3.1 Speedup over baseline code

Figure 8 shows the benefit of G-Opt for our microbenchmarks. G-Opt speeds up cuckoo hashing by 2.6x, pointer chasing (with $D = 100$) by 6.6x, and IPv6 lookups by 2x. The figure also shows the speedup obtained by manually re-arranging the baseline code to perform group prefetching. There is modest room for further optimization of the generated code in the future, but G-Opt performs surprisingly well compared to hand-optimized code: the manually optimized code is up to 5% faster than G-Opt. For every expensive memory access, G-Opt issues a prefetch, saves a label, and executes a `goto`, but the hand-optimized code avoids the last two steps.

3.3.2 Instruction overhead of G-Opt

G-Opt's output, $\mathcal{G}$, has more code than the original input function $\mathcal{F}$. The new function needs instructions to switch between different lookups, plus the initialization and termination code. G-Opt also replaces local variable accesses with array accesses. This can lead to additional load and store instructions because array locations are not register allocated.

Although G-Opt's code executes more instructions than the baseline code, *it uses fewer cycles* by reducing the number of cycles that are spent stalled on DRAM accesses. We quantify this effect in Figure 9 by measuring the total number of instructions and the instructions-per-cycle (IPC) for the baseline and with G-Opt. We use the PAPI tool [9] to access hardware counters for total retired instructions and total cycles. G-Opt offsets the increase in instruction count by an even larger increase in the IPC, leading to an overall decrease in execution time.

[2] This prefix length distribution is close to worst case; only 1.5% and 0.1% of real-world IPv6 prefixes are longer than 48 and 64 bits, respectively [14].

4 Evaluation

We evaluate four packet processing applications on CPUs and GPUs, each representing a different balance of computation, memory accesses, and overall processing required. We describe each application and list its computational and memory access requirements below. Although the CPU cycles used for packet header manipulation and transmission are an important source of computation, they are common to all evaluated applications and we therefore omit them from the per-application bullets. As described in Section 4.2, G-Opt also overlaps these computations with memory accesses.

Echo: To understand the limits of our hardware, we use a toy application called *Echo*. An Echo router forwards a packet to a uniformly random port P based on a random integer X in the packet's payload ($P = X \ mod \ 4$). In the GPU-offloaded version, we use the GPU to compute P from X. As this application does not involve expensive memory accesses, we do not use G-Opt on it.

IPv4 forwarding: We use Intel DPDK's implementation of the DIR-24-8-BASIC algorithm [22] for IPv4 lookups. It creates a 32 MB table for prefixes with length up to 24 bits and allocates 128 MB for longer prefixes. We populate the forwarding table with 527,961 prefixes from a BGP table snapshot [14], and use randomly generated IPv4 addresses in the workload. *Computation*: negligible. *Memory*: ~ 1 memory access on average (only 1% of our prefixes are longer than 24 bits).

IPv6 forwarding: As described in Section 3.3.

Layer-2 switch: We use the CuckooSwitch design [51]. It uses a cuckoo hash table to map MAC addresses to output ports. *Computation*: 1.5 hash-computations (on average) for determining the candidate buckets for a destination MAC address; comparing the destination MAC address with the addresses in the buckets' slots. *Memory*: 1.5 memory accesses (on average) for reading the buckets.

Named Data Networking: We use the hash-based algorithm for name lookup from Wang et al. [46], but use cuckoo hashing instead of their more complex perfect hashing scheme. We populate our name lookup table with prefixes from a URL dataset containing 10 million URLs [45, 46]. We make two simplifications for our GPU-accelerated NDN forwarding. First, because our hash function (`CityHash64`) is slow on the GPU, we use a *null kernel* that does not perform NDN lookups and returns a response immediately. Second, we use fixed-size 32-byte URLs (the average URL size used in Zhang et al. [49]) in the packet headers for both CPU and GPU, generated by randomly extending or truncating the URLs from the dataset.[3]

4.1 Experimental Setup

We conduct full-system experiments on a Xeon E5-2680 CPU (8 cores @2.70 GHz)-based server.[4] The CPU socket has 32 GB of quad-channel DDR3-1600 DRAM in its NUMA domain, 2 dual-port Intel X520 10 GbE NICs connected via PCIe 2.0 x8, and a GTX 980 connected via PCIe 2.0 x16. To generate the workload for the server, we use two client machines equipped with Intel L5640 CPUs (6 cores @2.27 GHz) and one Intel X520 NIC. The two 10 GbE ports on these machines are connected directly to two ports on the server. The machines run Ubuntu with Linux kernel 3.11.2 with Intel DPDK 1.5 and CUDA 6.0.

4.2 System design

Network I/O: We use Intel's DPDK [5] to access the NICs from userspace. We create as many RX and TX queues on the NIC ports as the number of active CPU cores, and ensure exclusive access to queues. Although the 40 Gbps of network bandwidth on the server machine corresponds to a maximum packet rate of 59.52 (14.88 * 4) million packets per second (Mpps) for minimum sized Ethernet frames, only 47.2 Mpps is achievable; the PCIe 2.0 x8 interface to the dual-port NIC is the bottleneck for minimum sized packets [51]. As the maximum gains from GPU acceleration come for small packets [23], we use the smallest possible packet size in all experiments.

GPU acceleration: We use PacketShader's approach to GPU-based packet processing as follows. We run a dedicated master thread that communicates with the GPU, and several worker threads that receive and transmit packets from the network. Using a single thread to communicate with the GPU is necessary because the overhead of CUDA functions increases drastically when called from multiple threads or processes. The worker threads extract the essential information from the packets and pass it on to the master thread using exclusive worker-master

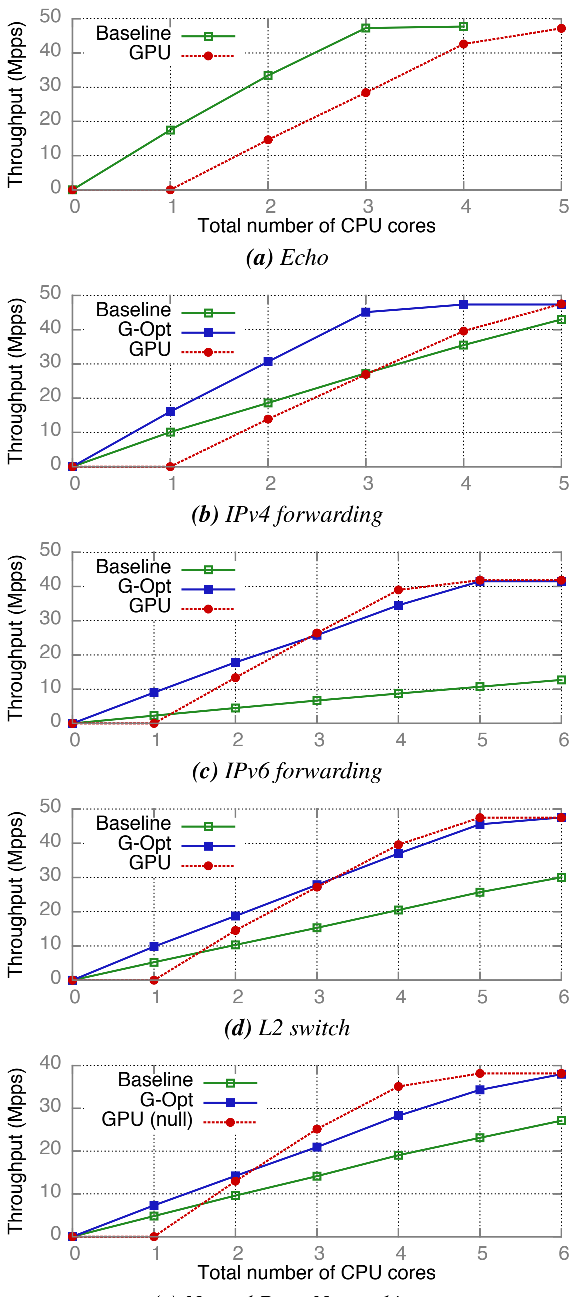


Figure 10: Full system throughput with increasing *total* CPU cores, N. For the GPU, N includes the master core (throughput is zero when $N = 1$ as there are no worker cores). The x-axis label is the same for all graphs.

queues. The workers also perform standard packet processing tasks like sanity checks and setting header fields. This division of labor between workers and master reduces the amount of data that the master needs to transmit to the GPU. For example, in IPv4 forwarding, the master receives only one 4-byte IPv4 address per received packet. In our implementation, each worker can have up to 4096 outstanding packets to the master.

[3]Our CPU version does not need to make these assumptions, and performs similarly with variable length URLs.

[4]The server is dual socket, but we restricted experiments to a single CPU to avoid noise from cross-socket QPI traffic. Previous work on software packet processing suggests that performance will scale and our results will apply to two socket systems [32, 51, 23].

PacketShader's master thread issues a separate CUDA `memcpy` for the data generated by each worker to transfer it to the GPU directly via DMA without first copying to the master's cache. Because of the large overhead of CUDA function calls (Figure 2), we chose not to use this approach.

Using G-Opt for packet processing programs: Intel DPDK provides functions to receive and transmit batches of packets. Using batching reduces function call and PCIe transaction overheads [23, 51] and is required for achieving the peak throughput. Our baseline code works as follows. First, it calls the batched receive function to get a batch of up to 16 packets from a NIC queue. It then passes this batch to the packet processing function $\mathcal{F}$, which processes the packets one by one.

We then apply G-Opt on $\mathcal{F}$ to generate the optimized function $\mathcal{G}$. Unlike the simpler benchmarks in Section 3.3, $\mathcal{F}$ is a full-fledged packet handler: it includes code for header manipulation and packet transmission in addition to the core data structure lookup. This gives $\mathcal{G}$ freedom to overlap the prefetches from the lookups with this additional code, but also gives it permission to transmit packets in a different order than they were received. However, $\mathcal{G}$ preserves the per-flow ordering if forwarding decisions are made based on packet headers only, as in all the applications above.[5] If so, all packets from the same flow are "switched out" by $\mathcal{G}$ at the same program points, ensuring that they reach the transmission code in order.

4.3 Workload generation

The performance of the above-mentioned packet processing applications depends significantly on two workload characteristics. The following discussion focuses on IPv4 forwarding, but similar factors exist for the other applications. First, the distribution of prefixes in the server's forwarding table, and the IP addresses in the workload packets generated by the clients, affects the cache hit rate in the server. Second, in real-world traffic, packets with the same IP address (e.g., from the same TCP connection) arrive in bursts, increasing the cache hit rate.

Although these considerations are important, recall that our primary focus is understanding the relative advantages of GPU acceleration as presented in previous work. We therefore tried to mimic PacketShader's experiments that measure the near worst-case performance of both CPUs and GPUs. Thus, for IPv4 forwarding, we used a real-world forwarding table and generated the IPv4 addresses in the packets with a uniform random distribution. For IPv6 forwarding, we populated the forwarding table with prefixes with randomly generated content, and chose the workload's addresses from these prefixes using uniformly random sampling.[6] We speculate that prior work may have favored these conditions because worst-case performance is an important factor in router design for quality of service and denial-of-service resilience. Based on results from previous studies [31, 48], we also expect that more cache-friendly (non-random) workloads are likely to improve CPU performance more than that of GPUs.

4.4 Throughput comparison

Figure 10 shows the throughput of CPU-only and GPU-accelerated software routers with different numbers of CPU cores. For Echo (Figure 10a), the CPU achieves ~ 17.5 Mpps of single-core throughput and needs 3 cores to saturate the 2 dual-port 10 GbE NICs. The GPU-offloaded implementation needs at least 4 worker cores, for a total of 5 CPU cores including the master thread. This happens because the overhead of communicating each request with the master reduces the single-worker throughput to 14.6 Mpps.

Figure 10b shows the graphs for IPv4 lookup. Without G-Opt, using a GPU provides some benefit: With a budget of 4 CPU cores, the GPU-accelerated version outperforms the baseline by 12.5%. After optimizing with G-Opt, the CPU version is strictly better than the GPU-accelerated version. G-Opt achieves the platform's peak throughput with 4 CPU cores, whereas the GPU-accelerated version requires 5 CPU cores *and* a GPU.

With G-Opt, a single core can process 16 million IPv4 packets per second, which is 59% higher than the baseline's single-core performance and is only 8.9% less than the 17.5 Mpps for Echos. When using the DIR-24-8-BASIC algorithm for IPv4 lookups, the CPU needs to perform only ~ 1 expensive memory access in addition to the work done in Echo. With G-Opt, the latency of this memory access for a packet is hidden behind independent packet-handling instructions from other packets. As GPUs also hide memory access latency, the GPU-accelerated version of IPv4 forwarding performs similarly to its Echo counterpart.

For IPv6 forwarding (Figure 10c), G-Opt increases single-core throughput by 3.8x from 2.2 Mpps to 8.4 Mpps. Interestingly, this increase is *larger* than G-Opt's 2x gain in local IPv6 lookup performance (Figure 8). This counter-intuitive observation is explained by the reduction in effectiveness of the reorder buffer for the baseline code: Due to additional packet handling instructions, the independent memory accesses for different packets in a batch are spaced farther apart in the forwarding code than in the local benchmarking code. These instructions consume slots in the processor's reorder buffer, reducing its ability to detect the inter-packet independence.

[5]For applications that also examine the packet content, the transmission code can be moved outside $\mathcal{F}$ for a small performance penalty.

[6]This workload is the worst case for DPDK's trie-based IPv6 lookup. PacketShader's IPv6 lookup algorithm uses hashing and shows worst-case behavior for IPv6 addresses with random content.

With G-Opt, our CPU-only implementation achieves 39 Mpps with 5 cores, and the platform's peak IPv6 throughput (42 Mpps) with 6 cores. Because IPv6 lookups require relatively heavyweight processing, our GPU-based implementation indeed provides higher *per-worker* throughput—it delivers line rate with only 4 worker cores, *but it requires another core for the master in addition to the GPU*. Therefore, using a GPU plus 5 CPU cores can provide a 7.7% throughput increase over using just 5 CPUs, but is equivalent to using 6 CPUs.

For the L2 switch (Figure 10d), G-Opt increases the throughput of the baseline by 86%, delivering 9.8 Mpps of single-core throughput. This is significantly smaller than the 17.5 Mpps for Echos because of the expensive hash computation required by cuckoo hashing. Our CPU-only implementation saturates the NICs with 6 cores, and achieves 96% of the peak throughput with 5 cores. In comparison, our GPU-accelerated L2 switch requires 5 CPU cores and a GPU for peak throughput.

For Named Data Networking, G-Opt increases single-core throughput from 4.8 Mpps to 7.3 Mpps, a 1.5x increase. With a budget of 4 CPU cores, the (simplified) GPU version's performance is 24% higher than G-Opt, but is almost identical if G-Opt is given one additional CPU core.

Conclusion: For all our applications, the throughput gain from adding a GPU is never larger than from adding just one CPU core. The cost of a Xeon E5-2680 v3 [6] core (more powerful than the cores used in this paper) is $150. In comparison, the cheapest GPU used in this paper costs $130 and consumes 110W of extra power. CPUs are therefore a more attractive and resource efficient choice than GPUs for these applications.

4.5 Latency comparison

The GPU-accelerated versions of the above applications not only require more resources than their G-Opt counterparts, but also add significant latency. Each round of communication with the GPU on our server takes ~ 20µs (Figure 2). As the packets that arrive during a round must wait for the next round to begin, the average latency added is $20 * 1.5 = 30$µs.

Our latency experiments measure the round-trip latency at clients. Ideally, we would have liked to measure the latency *added* by the server without including the latency added by the client's NIC and network stack. This requires the use of hardware-based traffic generators [42] to which we did not have access.[7]

In our experiments, clients add a timestamp to packets during transmission and use it to measure the RTT after reception. We control the load offered by clients by tuning the amount of time they sleep between packet transmissions. The large sleep time required for generating a low load, and buffered transmission at the server [32] cause our measured latency to be higher than our system's minimum RTT of 16µs.

For brevity, we present our latency-vs-throughput graphs only for Echo, and IPv4 and IPv6 forwarding. The CPU-only versions use G-Opt. All measurements used the minimum number of CPU cores required for saturating the network bandwidth.

Figure 11a shows that the RTT of CPU-only Echo is 29µs at peak throughput and 19.5µs at low load. The minimum RTT with GPU acceleration is 52µs, which is close to 30µs larger than the CPU-only version's minimum RTT. We observe similar numbers for IPv4 and IPv6 forwarding (Figures 11b and 11c), but the GPU version's latency increases at high load because of the larger batch sizes required for efficient memory latency hiding.

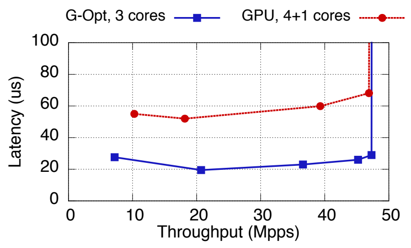
5 Discussion

Other similar optimizations for CPU programs Until now, we have discussed the benefit of an automatic DRAM latency-hiding optimization, G-Opt. We now discuss how intrusion detection systems (IDSes), *an application whose working set fits in cache* [41], can benefit from similar, latency-hiding optimizations.

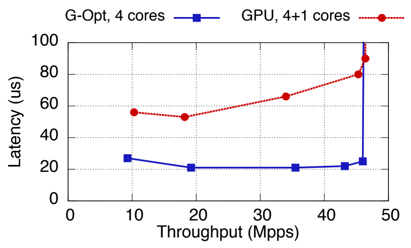
We study the packet filtering stage of Snort [39], a popular IDS. In this stage, each packet's payload is used to traverse one of several Aho-Corasick [15] DFAs. The DFA represents the set of malicious patterns against which this packet should be matched; Snort chooses which DFA to use based on the packet header. For our experiments, we recorded the patterns inserted by Snort v2.9.7 into its DFAs and used them to populate our simplified pattern matching engine. Our experiment uses 23,331 patterns inserted into 450 DFAs, leading to 301,857 DFA states. The workload is a `tcpdump` file from the DARPA Intrusion Detection Data Sets [11].

Our baseline implementation of packet filtering passes batches of B (~ 8) packets to a function that returns B lists of matched patterns. This function processes packets one-by-one. We made two optimizations to this function. First, we perform a loop interchange: Instead of completing one traversal before beginning another, we interweave them to give the CPU more independent instructions to reorder, reducing stalls from long-latency loads from cache. Second, we collect a larger batch of packets (8192 in our implementation), and sort it—first by the packet's DFA number and then by length. Sorting by DFA number reduces cache misses during batch traversal. Sorting by length increases the effectiveness of loop interchange—similar to minimizing control flow divergence for GPU-based traversals [41].

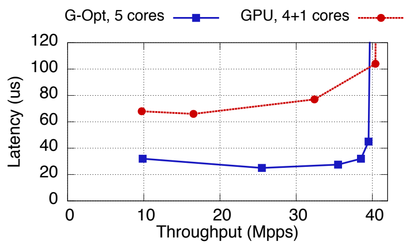
[7]Experiments with a Spirent SPT-N11U [42] traffic generator as the client have measured a minimum RTT of 7-8µs on an E5-2697 v2 server; the minimum RTT measured by our clients is 16µs.



(a) Echo



(b) IPv4 forwarding



(c) IPv6 forwarding

Figure 11: Full system latency with minimum CPU cores required to saturate network bandwidth.

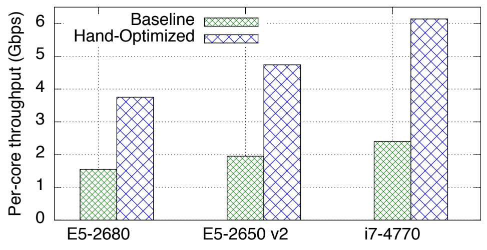


Figure 12: Pattern matching gains (no network I/O)

Figure 12 shows that, for a local experiment without network I/O, these optimizations increase single-core matching throughput by 2.4x or more. We believe that our optimizations also apply to the CPU-only versions of pattern matching in GPU-accelerated IDSes including Kargus [24] and Snap [43]. As we have only implemented the packet filtering stage (Snort uses a second, similar stage to discard false positives), we do not claim that CPUs can outperform GPUs for a full IDS. However, they can reduce the GPU advantage, or make CPU-only versions more cost effective. For example, in an experiment with innocent traffic, Kargus's throughput (with network I/O) improved between 1.4x and 2.4x with GPU offloading. Our pattern matching improvements offer similar gains which should persist in this experiment: innocent traffic rarely triggers the second stage, and network I/O requires less than 10% of the CPU cycles spent in pattern matching.

Additional applications We have shown that CPU implementations can be competitive with (or outperform) GPUs for a wide range of applications, including lightweight (IPv4 forwarding, Layer-2 switching), midweight (IPv6 and NDN forwarding), and heavyweight (intrusion detection) applications. Previous work explores the applicability of GPU acceleration to a number of different applications; one particularly important class, however, is cryptographic applications.

Cryptographic applications, on the one hand, involve large amounts of computation, making them seem attractive for vector processing [25, 23]. On the other hand, encryption and hashing requires copying the full packet data to the GPU (not just headers, for example). Since the publication of PacketShader, the first work in this area, Intel has implemented hardware AES encryption support for their CPUs. We therefore suspect that the 3.5x speedup observed in PacketShader for IPSec encryption would be unlikely to hold on today's CPUs. And, indeed, 6WIND's AES-NI-based IPSec implementation delivers 6 Gbps per core [1], 8x higher than PacketShader's CPU-only IPSec, though on different hardware.

One cryptographic workload where GPUs still have an advantage is processing expensive, but infrequent, RSA operations as done in SSLShader, assuming that connections arrive closely enough together for their RSA setup to be batched.[8] Being compute intensive, these cryptographic applications raise a second question for future work: Can automatic vectorization approaches (e.g., Intel's ISPC [8]) be used to increase the efficiency of CPU-based cryptographic applications?

Revising TCO estimates In light of the speedups we have shown possible for some CPU-based packet processing applications, it bears revisiting total-cost-of-ownership calculations for such machines. The TCO of a machine includes not just the cost of the CPUs, but the motherboard and chipset as well as the total system power draw, and the physical space occupied by the machine.

Although our measurements did not include power, several conclusions are obvious: Because the GPU-accelerated versions required almost as many CPU cores as the CPU-only versions, they are likely to use at least modestly more power than the CPU versions. The GTX 980 in our experiments can draw up to 165W compared to 130W for the E5-2680's 8 cores, though we lack precise power draw measurements.

Adding GPUs requires additional PCIe slots and lanes from the CPU, in addition to the cost of the GPUs. This burden is likely small for applications that require transferring only the packet header to the GPU, such as forwarding—but those applications are also a poor match for the GPU. It can, however, be significant for high-bandwidth offload applications, such as encryption and deep packet inspection.

[8]And perhaps HMAC-SHA1, but Intel's next generation "Skylake" CPUs will have hardware support for SHA-1 and SHA-256.

Future GPU trends may improve the picture. Several capabilities are on the horizon: CPU-integrated GPU functions may substantially reduce the cost of data and control transfers to the GPU. Newer NVidia GPUs support "GPUDirect" [7], which allows both the CPU and certain NICs to DMA directly packets to the GPU. GPUDirect could thus allow complete CPU-bypass from NIC to GPU, or reduce CUDA's overhead by letting the CPU write directly to GPU memory [29]. This technology currently has several restrictions—the software is nascent, and only expensive Tesla GPUs (over $1,700 each) and RDMA-capable NICs are supported. A more fundamental and long-term limitation of removing CPU involvement from packet processing is that it requires entire packets, not just headers, to be transferred to the GPU. The CPU's PCIe lanes would then have to be divided almost equally between NICs and GPUs, possibly halving the network bandwidth that the system can handle.

Alternative architectures such as Tilera's manycore designs, which place over 100 cores on a single chip with high I/O and memory bandwidth, or Intel's Xeon Phi, are interesting and under-explored possibilities. Although our results say nothing about the relative efficiency of these architectures, we hope that our techniques will enable better comparisons between them and traditional CPUs.

Handling updates Currently, G-Opt works only for data structures that are not updated concurrently. This constraint also applies to GPU-accelerated routers where the CPU constructs the data structure and ships it to the GPU. It is possible to hide DRAM latency for updates using manual group prefetching [32]; if updates are relatively infrequent, they also can be handled outside the batch lookup code. Incorporating updates into G-Opt is part of future work.

6 Related Work

GPU-based packet processing Several systems have used GPUs for IPv4 lookups *absent* network I/O [50, 31, 30, 35], demonstrating substantial speedups. Our end-to-end measurements that include network I/O, however, show that there is very little room for improving IPv4 lookup performance—when IPv4 forwarding is optimized with G-Opt, the single-core throughput drops by less than 9% relative to Echo. Packet classification requires matching packet headers against a corpus of rules; the large amount of per-packet processing makes it promising for GPU acceleration [23, 27, 44]. GSwitch [44] is a recent GPU-accelerated packet classification system. We believe that the Bloom filter and hash table lookups in GSwitch's CPU version can benefit from G-Opt's latency hiding, reducing the GPU's advantage.

CPU-based packet processing RouteBricks [18] focused on mechanisms to allocate packets to cores; its techniques are now standard for making effective use of a multicore CPU for network packet handling. User-level networking frameworks like Intel's DPDK [5], `netmap` [38], and PF_RING [10] provide a modern and efficient software basis for packet forwarding, which our work and others takes advantage of. Many of the insights in this paper were motivated by our prior work on hiding lookup latency in CuckooSwitch [51], an L2 switch that achieves 80 Gbps while storing a billion MAC addresses.

Hiding DRAM latency for CPU programs is important in many contexts: Group prefetching and software pipelining has been used to mask DRAM latency for database hash-joins [17], a software-based L2 switch [51], in-memory trees [40, 28], and in-memory key-value stores [32, 34, 26]. These systems required manual code rewrites. To our knowledge, G-Opt is the first method to automatically hide DRAM latency for the independent lookups in these applications.

7 Conclusion

Our work challenges the conclusions of prior studies about the relative performance advantages of GPUs in packet processing. GPUs achieve their parallelism and performance benefits by constraining the code that programmers can write, but this very coding paradigm also allows for latency-hiding CPU implementations. Our G-Opt tool provides a semi-automated way to produce such implementations. CPU-only implementations of IPv4, IPv6, NDN, and Layer-2 forwarding can thereby be more resource efficient and add lower latency than GPU implementations. We hope that enabling researchers and developers to more easily optimize their CPU-based designs will help improve future evaluation of both hardware and software-based approaches for packet processing. Although we have examined a wide range of applications, this work is not the end of the line. Numerous other applications have been proposed for GPU-based acceleration, and we believe that these techniques may be applicable to other domains that involve read-mostly, parallelizable processing of small requests.

Code release The code for G-Opt and the experiments in this paper is available at `https://github.com/efficient/gopt`.

Acknowledgements This work was supported by funding from the National Science Foundation under awards CNS-1314721, CCF-0964474, and 1345305; and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC). Emulab [47] and PRObE [21] were used for some experiments. PRObE is supported in part by NSF awards CNS-1042537 and CNS-1042543 (PRObE). We thank Sangjin Han and Hyeontaek Lim for valuable comments, and David Maltz for shepherding.

References

[1] High-Performance Packet Processing Solutions for Intel Architecture Platforms. http://www.lannerinc.com/downloads/campaigns/LIDS/05-LIDS-Presentation-Charlie-Ashton-6WIND.pdf.

[2] ANTLR. http://www.antlr.org.

[3] NVIDIA CUDA. http://www.nvidia.com/object/cuda_home_new.html.

[4] CityHash. https://code.google.com/p/cityhash/.

[5] Intel DPDK: Data Plane Development Kit. http://dpdk.org.

[6] Intel Xeon Processor E5-2680 v3. http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz.

[7] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect.

[8] Intel's SPMD Program Compiler. https://ispc.github.io.

[9] Performance Application Programming Interface (PAPI). http://icl.cs.utk.edu/papi/.

[10] PF_RING: High-speed packet capture, filtering and analysis. http://www.ntop.org/products/pf_ring/.

[11] DARPA Intrusion Detection Data Sets, . http://www.ll.mit.edu/mission/communications/cyber/CSTcorpora/ideval/data/.

[12] NetFPGA, . http://yuba.stanford.edu/NetFPGA/.

[13] Open vSwitch, . http://www.openvswitch.org.

[14] University of Oregon Route Views Project, . http://www.routeviews.org.

[15] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, June 1975.

[16] J.-P. Aumasson and D. J. Bernstein. SipHash: a fast short-input PRF. In *INDOCRYPT*, 2012.

[17] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving Hash Join Performance Through Prefetching. *ACM Trans. Database Syst. 2007*.

[18] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *SOSP 2009*.

[19] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *NSDI*, 2013.

[20] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud Scale Load Balancing with Hardware and Software. In *SIGCOMM 2014*.

[21] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. PRObE: A Thousand-Node Experimental Cluster for Computer Systems Research.

[22] P. Gupta, S. Lin, and M. Nick. Routing Lookups in Hardware at Memory Access Speeds. In *INFOCOM 1998*.

[23] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *SIGCOMM 2010*.

[24] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: A Highly-scalable Software-based Intrusion Detection System. In *CCS 2012*.

[25] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *NSDI 2011*.

[26] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *SIGCOMM*, 2014.

[27] K. Kang and Y. S. Deng. Scalable Packet Classification via GPU Metaprogramming. In *DATE*, 2011.

[28] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Designing Fast Architecture-sensitive Tree Search on Modern Multicore/Many-core Processors. *ACM TODS 2011*, .

[29] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *OSDI 2014*, .

[30] T. Li, H. Chu, and P. Wang. IP Address Lookup Using GPU. In *HPSR*, 2013.

[31] Y. Li, D. Zhang, A. X. Liu, and J. Zheng. GAMT: A Fast and Scalable IP Lookup Engine for GPU-based Software Routers. In *ANCS 2013*.

[32] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *USENIX NSDI*, 2014.

[33] D. Lustig and M. Martonosi. Reducing GPU Offload Latency via Fine-grained CPU-GPU Synchronization. In *HPCA 2013*.

[34] Y. Mao, C. Cutler, and R. Morris. Optimizing RAM-latency Dominated Applications. In *APSys 2013*.

[35] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng, and S. Zhang. IP Routing Processing with Graphic Processors. In *DATE*, 2010.

[36] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Grappa: A Latency-Tolerant Runtime for Large-Scale Irregular Applications. Technical Report UW-CSE-14-02-01, University of Washington.

[37] R. Pagh and F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, May 2004.

[38] L. Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC 2012*.

[39] M. Roesch and S. Telecommunications. Snort - Lightweight Intrusion Detection for Networks. 1999.

[40] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *PVLDB 2011*.

[41] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for Network Packet Signature Matching. In *ISPASS*, 2009.

[42] Spirent. Spirent SPT-N11U. http://www.spirent.com/sitecore/content/Home/Ethernet_Testing/

Platforms/11U_Chassis.

[43] W. Sun and R. Ricci. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In *ANCS 2013*.

[44] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman. Multi-Layer Packet Classification with Graphics Processing Units. In *CoNEXT*, 2014.

[45] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, H. Wu, and D. Yang. Wire Speed Name Lookup: A GPU-based Approach. In *NSDI 2013*.

[46] Y. Wang, B. Xu, D. Tai, J. Lu, T. Zhang, H. Dai, B. Zhang, and B. Liu. Fast name lookup for Named Data Networking. In *IWQoS*, 2014.

[47] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI*, 2002.

[48] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy. Guarantee IP Lookup Performance with FIB Explosion. In *SIGCOMM*, 2014.

[49] T. Zhang, Y. Wang, T. Yang, J. Lu, and B. Liu. NDNBench: A benchmark for Named Data Networking lookup. In *GLOBECOM*, 2013.

[50] J. Zhao, X. Zhang, X. Wang, Y. Deng, and X. Fu. Exploiting Graphics Processors for High-performance IP Lookup in Software Routers. *INFOCOM*, 2011.

[51] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *CoNEXT*, 2013.