Harnessing GPU Computing in System Level Software

Weibin Sun
Software stack needs fast system level base!
Software stack needs fast system level base!

System functionality can be expensive!
Software stack needs fast system level base!

System functionality can be expensive!

- **Crypto**: AES, SHA1, MD5, RSA
- **Lookup**: IP routing, DNS, key-value store, DB
- **Erasure coding**: RAID, distributed storage, wireless
- **Pattern matching**: NIDS, virus, searching
- **Compression**: storage, network
- ...
Software stack needs fast system level base!

System functionality can be expensive!
Software stack needs fast system level base!

System functionality can be expensive!

Modern digital workloads are big!
Software stack needs fast system level base!

System functionality can be expensive!

Modern digital workloads are big!

System software needs high throughput processing!
Software stack needs fast system level base!

System functionality can be expensive!

Modern digital workloads are big!

System software needs high throughput processing!
Software stack needs fast system level base!

System functionality can be expensive!

Modern digital workloads are big!

System software needs high throughput processing!
Software stack needs fast system level base!

System functionality can be expensive!

Modern digital workloads are big!

System software needs high throughput processing!
Many system tasks are data-parallelizable!
Many system tasks are data-parallelizable!

packets, sectors, data blocks, …
Many system tasks are data-parallelizable!
Many system tasks are data-parallelizable!

GPUs are the de-facto standard parallel processors!
Many system tasks are data-parallelizable!

GPUs are the de-facto standard parallel processors!
Software stack needs fast system level base!

System software needs high throughput processing!

System functionality can be expensive!

Modern digital workloads are big!

Many system tasks are data-parallelizable!

GPUs are the de-facto standard parallel processors!
Many system tasks are data-parallelizable!

Using GPUs in system software!

GPUs are the de-facto standard parallel processors!
Thesis statement

The throughput of system software with parallelizable, computationally expensive tasks can be improved by using GPUs and frameworks with memory efficient and throughput oriented designs.
Thesis statement

The throughput of system software with parallelizable, computationally expensive tasks can be improved by using GPUs and frameworks with memory efficient and throughput oriented designs.
Thesis statement

The throughput of system software with parallelizable, computationally expensive tasks can be improved by using GPUs and frameworks with memory efficient and throughput oriented designs.
Thesis statement

The throughput of system software with parallelizable, computationally expensive tasks can be improved by using GPUs and frameworks with memory efficient and throughput oriented designs.
The throughput of system software with parallelizable, computationally expensive tasks can be improved by using GPUs and frameworks with memory efficient and throughput oriented designs.
Thesis work

• Generic principles

• Concretely:
  • Two generic frameworks for representative system tasks: storage and network
  • Example applications on top of both frameworks

• Literature survey *
GPU essentials
CUDA GPU essentials
CUDA GPU essentials

SIMT wimpy cores: 32-core warp
CUDA GPU essentials

SIMT wimpy cores: 32-core warp  wide memory interface: 384bits!
CUDA GPU essentials

SIMT wimpy cores:
32-core warp

wide memory interface: 384bits!

necessary memory copy
Related work: pioneers

- Gnort [RAID ’2008]
- PacketShader [SIGCOMM ’2010]
- SSLShader [NSDI ’2011]
- EigenCFA [POPL ’2011]
- Gibraltar [ICPP ’2010]
- CrystalGPU [HPDC ’2010]
- ...
Related work: pioneers

- Gnort [RAID ’2008]
- PacketShader [SIGCOMM ’2010]
- SSLShader [NSDI ’2011]
- EigenCFA [POPL ’2011]
- Gibraltar [ICPP ’2010]
- CrystalGPU [HPDC ’2010]
- ...

Specialized and Ad-hoc solutions to one particular application!
GPU computing frameworks/models

- Hydra [ASPLOS’08]
- CUDA-Lite [LCPC’08]
- hiCUDA [GPGPU’09]
- ASDSM [ASPLOS’10]
- Sponge [ASPLOS’11]
- CGCM [PLDI’11]
- PTask [SOSP’11]
- …
GPU computing frameworks/models

- Hydra [ASPLOS’08]
- CUDA-Lite [LCPC’08]
- hiCUDA [GPGPU’09]
- ASDSM [ASPLOS’10]
- Sponge [ASPLOS’11]
- CGCM [PLDI’11]
- PTask [SOSP’11]
- …

- Simple workflow
- Long computation, PCIe copy and synchronization not matter
- No batched data divergence
GPU computing frameworks/models

- Hydra [ASPLOS’08]
- CUDA-Lite [LCPC’08]
- hiCUDA [GPGPU’09]
- ASDSM [ASPLOS’10]
- Sponge [ASPLOS’11]
- CGCM [PLDI’11]
- PTask [SOSP’11]
- ...

System level GPU computing needs its own generic frameworks!
Rest of this talk

• Generic principles
• GPUstore : for storage systems
• Snap : for packet processing
System level GPU computing: generic principles
Problems

- Latency-oriented system code V.S. throughput-oriented GPU
- CPU-GPU synchronization hurts asynchronous systems
- Wasted PCIe data transfer
- Double buffering for GPU DMA
Problems

- Latency-oriented system code V.S. throughput-oriented GPU
- CPU-GPU synchronization hurts asynchronous systems
- Wasted PCIe data transfer
- Double buffering for GPU DMA
Latency V.S. throughput
Latency V.S. throughput

Batched processing: wait for enough workload
Latency V.S. throughput

Batched processing: wait for enough workload

• how much is enough?
Latency V.S. throughput

Batched processing: wait for enough workload
• how much is enough?
• increased latency
Problems

- Latency-oriented system code V.S. throughput-oriented GPU
- CPU-GPU synchronization hurts asynchronous systems
- Wasted PCIe data transfer
- Double buffering for GPU DMA
Problems

- Batched processing
- CPU-GPU synchronization hurts asynchronous systems
- Wasted PCIe data transfer
- Double buffering for GPU DMA
Problems

- Batched processing
- CPU-GPU synchronization hurts asynchronous systems
- Wasted PCIe data transfer
- Double buffering for GPU DMA
CPU-GPU synchronization

Blocking:

CPU: 

GPU:
CPU-GPU synchronization

Blocking:

CPU: 

GPU:
CPU-GPU synchronization

Blocking:

CPU:  

GPU:  

---
CPU-GPU synchronization

Blocking:

CPU: [Diagram showing CPU states]

GPU: [Diagram showing GPU states]

sleep for sync
CPU-GPU synchronization

Blocking:

CPU:

GPU:

sleep for sync

Non-blocking:

CPU:

GPU:
CPU-GPU synchronization

**Blocking:**

- CPU: **sleep for sync**
- GPU:

**Non-blocking:**

- CPU: **keep working**
- GPU:
CPU-GPU synchronization

Blocking:

System code (network, disk, etc.) likes:

- Callback
- Polling

Non-blocking:
Problems

- Batched processing
- CPU-GPU synchronization hurts asynchronous systems
- Wasted PCIe data transfer
- Double buffering for GPU DMA
Problems

- Batched processing
- Asynchronous non-blocking GPU programming
- Wasted PCIe data transfer
- Double buffering for GPU DMA
Problems

- Batched processing
- Asynchronous non-blocking GPU programming
- Wasted PCIe data transfer
- Double buffering for GPU DMA
How PCIe transfer wasted?
How PCIe transfer wasted?

Do you need the entire packet to do routing lookup?
How PCIe transfer wasted?

Do you need the entire packet to do routing lookup?

What you need:
How PCIe transfer wasted?

Do you need the entire packet to do routing lookup?

What you need:

4 bytes
How PCIe transfer wasted?

Do you need the entire packet to do routing lookup?

What you need:

4 bytes

What you may transfer:
How PCIe transfer wasted?

Do you need the entire packet to do routing lookup?

What you need:

What you may transfer:

\[ \geq 64 \text{ bytes} \]
Compact your workload data

For ``use-partial`` data usage pattern
Problems

- Batched processing
- Asynchronous non-blocking GPU programming
- Wasted PCIe data transfer
- Double buffering for GPU DMA
Problems

- Batched processing
- Asynchronous non-blocking GPU programming
- Reduce PCIe data transfer with compacted workload
- Double buffering for GPU DMA
Problems

- Batched processing
- Asynchronous non-blocking GPU programming
- Reduce PCIe data transfer with compacted workload
- Double buffering for GPU DMA
GPU DMA contraints
GPU DMA contraints

- DMA only on locked memory pages
GPU DMA constraints

- DMA only on locked memory pages
- GPU drivers only recognize their own locked memory pool
GPU DMA constraints

- DMA only on locked memory pages
- GPU drivers only recognize their own locked memory pool
GPU DMA constraints

- DMA only on locked memory pages
- GPU drivers only recognize their own locked memory pool
GPU DMA constraints

- DMA only on locked memory pages
- GPU driver’s memory area

Make GPU driver and system code use the same locked memory for zero-copy DMA!
Problems

- Batched processing
- Asynchronous non-blocking GPU programming
- Reduce PCIe data transfer with compacted workload
- Double buffering for GPU DMA
Problems

- Batched processing
- Asynchronous non-blocking GPU programming
- Reduce PCIe data transfer with compacted workload
- All use the same locked memory for zero-copy DMA
Latency-oriented system code VS throughput-oriented GPU

- Batched processing
- Asynchronous non-blocking GPU programming
- Reduce PCIe data transfer with compacted workload
- All use the same locked memory for zero-copy DMA
GPUstore: Harnessing GPU Computing in OS Storage Systems

[Weibin Sun, Robert Ricci, Matthew L. Curry @ SYSTOR’12]
GPUstore overview
GPUstore overview

• A storage framework for Linux kernel to use CUDA GPUs in filesystems, block drivers, etc.
GPUstore overview

- A storage framework for Linux kernel to use CUDA GPUs in filesystems, block drivers, etc.
GPUstore overview

- A storage framework for Linux kernel to use CUDA GPUs in filesystems, block drivers, etc.
- Minimally invasive GPU integration
GPUstore overview

- A storage framework for Linux kernel to use CUDA GPUs in filesystems, block drivers, etc.
- Minimally invasive GPU integration
  - Small changes
GPUstore overview

• A storage framework for Linux kernel to use CUDA GPUs in filesystems, block drivers, etc.

• Minimally invasive GPU integration
  • Small changes
  • Preserve interface and semantics
GPUstore overview

- A storage framework for Linux kernel to use CUDA GPUs in filesystems, block drivers, etc.
- Minimally invasive GPU integration
  - Small changes
  - Preserve interface and semantics
  - Keep efficient
GPUstore integration

- Syscall
- VFS
- Filesystems... e.g. eCryptfs
- Page Cache
- Functional virtual block drivers... e.g. dm-crypt, MD RAID6
- Block device drivers
GPUstore integration

- Syscall
  - VFS
    - Filesystems... e.g. eCryptfs
      - Page Cache
        - Functional virtual block drivers... e.g. dm-crypt, MD RAID6
          - Block device drivers
GPUstore integration

- Syscall
- VFS
- Filesystems… e.g. eCryptfs
- Page Cache
- Functional virtual block drivers… e.g. dm-crypt, MD RAID6
- Block device drivers

GPUCipher(buf);

GPURSCode(buf);
GPUstore integration

- Syscall
- VFS
- Filesystems... e.g. eCryptfs
- Page Cache
- Functional virtual block drivers... e.g. dm-crypt, MD RAID6
- Block device drivers

Computation Requests

GPU Services
- GPU Ciphers
- GPU RAID6

GPUCipher(buf);

GPURSCode(buf);
GPUstore request workflow

1. sys-calls
2. Service Users ...
3. Service-specific Scheduling
4. Execution Preparation
5. GPU Execution
6. Post-execution
7. CPU
8. GPU
GPUstore request workflow

sys-calls

Service Users...

requests

Service-specific Scheduling

Execution Preparation

Post-execution

GPU Execution

Service interface

CPU

GPU
Apply generic principles

- Batched processing
- Asynchronous non-blocking GPU programming
- Reduce PCIe data transfer with compacted workload
- All use the same locked memory for zero-copy DMA
Apply generic principles

- Batched processing
- Asynchronous non-blocking GPU programming
Apply generic principles

- Batched processing
- Merge small requests
- Asynchronous non-blocking GPU programming
Apply generic principles

- Batched processing
- Merge small requests
- Asynchronous non-blocking GPU programming
- Split large requests
Apply generic principles

- Batched processing
- Merge small requests
- Asynchronous non-blocking GPU programming
- Split large requests
- Callback-based request processing
Request scheduling

- Merge small requests
- Split large requests
Request scheduling

- Merge small requests
- Split large requests

- How to define “small” and “large”?
Request scheduling

- Merge small requests
- Split large requests

- How to define “small” and “large”?
- Up to specific services.
Apply generic principles

- Batched processing
- Asynchronous non-blocking GPU programming
- Reduce PCIe data transfer with compacted workload
- All use the same locked memory for zero-copy DMA
Apply generic principles

- Reduce PCIe data transfer with compacted workload
- All use the same locked memory for zero-copy DMA
Apply generic principles

- All use the same locked memory for zero-copy DMA
Apply generic principles

- All use the same locked memory for zero-copy DMA
- User space CUDA library dependency
Apply generic principles

- All use the same locked memory for zero-copy DMA
  - User space CUDA library dependency
  - In-kernel CUDA’s locked memory allocator
In-kernel CUDA locked memory allocator

CUDA mem
continuous VA

User space (helper)
Kernel space

Physical pages

Continuous VA

Kernel vmap
What's next?

Replace `kmalloc()`/`vmalloc()`/`get_free_pages()` with `GPUstoreMalloc()`?
In-kernel CUDA locked memory allocator

What’s next?
Replace `kmalloc() / vmalloc() / get_free_pages()` with `GPUstoreMalloc()`?

Sometimes it is infeasible!
Why infeasible?
Why infeasible?

- Syscall
- VFS
- Filesystem
- Page cache
- Block I/O
- Virtual Block Driver
- Block Driver
Why infeasible?

- Who allocates the memory?
Why infeasible?

- Who allocates the memory?
  - “pass-by-ref” interface
Why infeasible?

- Who allocates the memory?
  - "pass-by-ref" interface

- Can NOT modify highly-depended cache allocators:
Why infeasible?

- Who allocates the memory?
  - “pass-by-ref” interface

- Can NOT modify highly-dependied cache allocators:
  - page cache, buffer cache
  - object cache
  - packet pool
  - ...

Diagram:
- Syscall
- VFS
- Filesystem
- Page cache
- Block I/O
- Virtual Block Driver
- Block Driver
Why infeasible?

- Who allocates the memory?

Can we use arbitrary memory for GPU DMA?
Remap external memory for GPU DMA
Remap external memory for GPU DMA
Remap external memory for GPU DMA

GPU driver’s locked memory area (vma)
Remap external memory for GPU DMA

GPU driver’s locked memory area (vma)

External mem

set page-table entries

DMA locked mem

GPU mem
Remap external memory for GPU DMA

GPU driver’s locked memory area (vma)

External mem

set page-table entries

DMA locked mem

DMA on PCIe

GPU mem
Apply generic principles

- All use the same locked memory for zero-copy DMA
Apply generic principles

- All use the same locked memory for zero-copy DMA
- In-kernel CUDA locked memory allocator
Apply generic principles

- All use the same locked memory for zero-copy DMA
- In-kernel CUDA locked memory allocator
- Remap external memory into GPU driver’s locked memory area
Implementation and evaluation
Implementation and evaluation

• Case studies of major storage residents:
  • **dm-crypt**: disk encryption layer
  • **eCryptfs**: encrypted filesystem
  • **md**: software RAID6
Implementation and evaluation

• Case studies of major storage residents:
  • `dm-crypt`: disk encryption layer
  • `eCryptfs`: encrypted filesystem
  • `md`: software RAID6

• Integration cost:

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Total LOC</th>
<th>Modified LOC</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>dm-crypt</td>
<td>1,800</td>
<td>50</td>
<td>3%</td>
</tr>
<tr>
<td>eCryptfs</td>
<td>11,000</td>
<td>200</td>
<td>2%</td>
</tr>
<tr>
<td>md</td>
<td>6,000</td>
<td>20</td>
<td>0.3%</td>
</tr>
</tbody>
</table>
AES cipher performance

Throughput (MB/s)

Buffer Size

- CPU
- Base GPU
AES cipher performance

Throughput (MB/s)

Buffer Size

4GB 8GB 16GB 32GB 64GB 128GB 256GB 512GB 1MB 2MB 4MB

CPU

Base GPU

GPU with Split
AES cipher performance

Throughput (MB/s)

Buffer Size

4KB 8KB 16KB 32KB 64KB 128KB 256KB 512KB 1MB 2MB 4MB

CPU
Base GPU
GPU with Split
GPU no RB
AES cipher performance

![Graph showing AES cipher performance with different buffer sizes and throughput measurements.]
Faster than SSD: dm-crypt

Throughput (MB/s)

Read/Write size

GPU Read
GPU Write
CPU Read
CPU Write

Figure 4. GPU AES cipher throughput with different optimizations compared with Linux kernel’s CPU implementation. The experiments marked “w/o RB” use the techniques described in Section 2.2.2 to avoid redundant buffering.

Figure 5. dm-crypt throughput on an SSD-backed device. Our second microbenchmark shows the effects of our optimization to remove redundant buffering and the split operation. This benchmark, also run on S1, uses the AES cipher service on the GPU, and the results can be seen in Figure 4. The baseline GPU result shows a speedup over the CPU cipher, demonstrating the feasibility of GPU acceleration for such computation. Our split operation doubles performance at large block sizes, and eliminating redundant buffering triples performance at sizes of 256 KB or larger. Together, these two optimizations give a speedup of approximately four times, and with them, the GPU-accelerated AES cipher achieves a speedup of 36 times over the CPU AES implementation in the Linux kernel. The performance levels approach those seen in Figure 3, implying that the memory copy, rather than the AES cipher computation, is the bottleneck.

4.2 dm-crypt Sequential I/O

Next, we use the dd tool to measure raw sequential I/O speed in dm-crypt. The results shown in Figure 5 indicate that with read and write sizes of about 1MB or larger, the GPU-accelerated dm-crypt easily reaches our SSD’s maximum throughput (250MB/s read and 170MB/s write). The CPU version is 60% slower; while it would be fast enough to keep up with a mechanical hard disk, it is unable reach the full potential of the SSD. Substituting a RAM disk for the SSD (Figure 6), we see that the GPU-accelerated dm-crypt was limited by the speed of the drive: it is able to achieve a maximum read throughput of 1.4 GB/s, more than six times as fast as the CPU implementation. This is almost exactly the rated read speed for the ioDrive Duo, currently the third fastest SSD in production [11]. As the throughput of storage systems rises, GPUs present a promising way to place computation into those systems while taking full advantage of the speed of the underlying storage devices.

4.3 eCryptfs Sequential and Concurrent Access

Figure 7 and Figure 8 compare the sequential performance for the CPU and GPU implementation of eCryptfs. We used the iozone tool to do sequential reads and writes using varying block sizes and measured the resulting throughput. Because eCryptfs does not support direct I/O, effects from kernel features such as the page cache and readahead affect our results. To minimize (but not completely eliminate) these effects, we cleared the page cache before running read-only benchmarks, and all writes were done synchronously.

Figure 7 shows that on the SSD, the GPU achieves 250 MBps when reading, compared with about 150 MBps...
Faster than SSD: dm-crypt

Figure 4. GPU AES cipher throughput with different optimizations compared with Linux kernel's CPU implementation. The experiments marked "w/o RB" use the techniques described in Section 2.2.2 to avoid redundant buffering.

Our second microbenchmark shows the effects of our optimization to remove redundant buffering and the split operation. This benchmark, also run on S1, uses the AES cipher service on the GPU, and the results can be seen in Figure 4. The baseline GPU result shows a speedup over the CPU cipher, demonstrating the feasibility of GPU acceleration for such computation. Our split operation doubles performance at large block sizes, and eliminating redundant buffering triples performance at sizes of 256 KB or larger. Together, these two optimizations give a speedup of approximately four times, and with them, the GPU-accelerated AES cipher achieves a speedup of 36 times over the CPU AES implementation in the Linux kernel. The performance levels approach those seen in Figure 3, implying that the memory copy, rather than the AES cipher computation, is the bottleneck.

4.2 dm-crypt Sequential I/O

Next, we use the dd tool to measure raw sequential I/O speed in dm-crypt. The results shown in Figure 5 indicate that with read and write sizes of about 1MB or larger, the GPU-accelerated dm-crypt easily reaches our SSD's maximum throughput (250MB/s read and 170MB/s write). The CPU version is 60% slower; while it would be fast enough to keep up with a mechanical hard disk, it is unable reach the full potential of the SSD. Substituting a RAM disk for the SSD (Figure 6), we see that the GPU-accelerated dm-crypt was limited by the speed of the drive: it is able to achieve a maximum read throughput of 1.4 GB/s, more than six times as fast as the CPU implementation. This is almost exactly the rated read speed for the ioDrive Duo, currently the third fastest SSD in production [11]. As the throughput of storage systems rises, GPUs present a promising way to place computation into those systems while taking full advantage of the speed of the underlying storage devices.

4.3 eCryptfs Sequential and Concurrent Access

Figure 7 and Figure 8 compare the sequential performance for the CPU and GPU implementation of eCryptfs. We used the iozone tool to do sequential reads and writes using varying block sizes and measured the resulting throughput. Because eCryptfs does not support direct I/O, effects from kernel features such as the page cache and readahead affect our results. To minimize (but not completely eliminate) these effects, we cleared the page cache before running read-only benchmarks, and all writes were done synchronously. Figure 7 shows that on the SSD, the GPU achieves 250 MBps when reading, compared with about 150 MBps SSD: 250MB/s.
Working with existing optimization

eCryptfs on RAM disks

Linux max 128KB read-ahead effects on read
GPUstore Summary

- Enables efficient GPU-accelerated storage in Linux kernel with small changes

- https://github.com/wbsun/kgpu

- Details not presented: user-space helper, non-blocking K-U comm, more exp result
Snap: Fast and Flexible Packet Processing With GPUs and Click

[Weibin Sun, Robert Ricci @ ANCS’13]
Click background [Kohler, et.al. TOCS’00]

- Elements
- Ports
  - push/pull
Why using GPUs in Click?

<table>
<thead>
<tr>
<th>Configurations</th>
<th>Throughput</th>
<th>Slow down</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Forwarder: FromDevice (ethX, RingY) -ToDevice (ethX, RingY)</td>
<td>30.97 Gbps</td>
<td>N/A</td>
</tr>
<tr>
<td>Simple SDN Forwarder: FromDevice (ethX, RingY) -SDNClassifier -ToDevice (ethX, RingY)</td>
<td>17.7 Gbps</td>
<td>42.7%</td>
</tr>
</tbody>
</table>

Why using GPUs in Click?

Configurations

- **Simple Forwarder:**
  - FromDevice (ethX, RingY)
  - ToDevice (ethX, RingY)

- **Simple SDN Forwarder:**
  - FromDevice (ethX, RingY)
  - SDNClassifier
  - ToDevice (ethX, RingY)
Why using GPUs in Click?

Configurations

<table>
<thead>
<tr>
<th>Simple Forwarder (ethX, RingY)</th>
<th>Throughput</th>
<th>Slow down</th>
</tr>
</thead>
<tbody>
<tr>
<td>FromDevice (ethX, RingY)</td>
<td>30.97 Gbps</td>
<td>N/A</td>
</tr>
<tr>
<td>ToDevice (ethX, RingY)</td>
<td>17.7 Gbps</td>
<td>42.7%</td>
</tr>
</tbody>
</table>

Packet processing needs more computing power!
Why using GPUs in Click?
Why using GPUs in Click?

(a) IP lookup algorithm (radix tree)
Why using GPUs in Click?

And the GPU rocks!
Snap: the idea

- Moving parts of Click pipeline onto GPUs
- CPU for sequential
- GPU for parallel
- Keep Click’s modular style
Apply generic principles

- Batched processing
- Asynchronous non-blocking GPU programming
- Reduce PCIe data transfer with compacted workload
- All use the same locked memory for zero-copy DMA
Apply generic principles

- Batched processing
Snap batched processing
Snap batched processing

Single Packet Path

push/pull  Packet

Element
+ push()
+ pull()
Snap batched processing

- Single Packet Path
  - push/pull
  - Packet
  - Batcher

- Multi-Packet Path
  - bpush/bpull
  - PacketBatch

- Element
  - + push()
  - + pull()

- BEElement
  - + bpush()
  - + bpull()
Snap batched processing

Single Packet Path

**push/pull** Packet

Batcher

**bpush/bpull** PacketBatch

Multi-Packet Path

Debatcher

Single Packet Path

<table>
<thead>
<tr>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ push()</td>
</tr>
<tr>
<td>+ pull()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ bpush()</td>
</tr>
<tr>
<td>+ bpull()</td>
</tr>
</tbody>
</table>
Batched packet divergence

- Packets in a batch go to different paths
Batched packet divergence

- Packets in a batch go to different paths
Batched packet divergence

- Packets in a batch go to different paths

Two choices:
Batched packet divergence

- Packets in a batch go to different paths

Two choices:
1. Split the batch into two copies
Batched packet divergence

- Packets in a batch go to different paths

Two choices:
1. Split the batch into two copies
   - needs PCIe copy, sync!
Batched packet divergence

• Packets in a batch go to different paths

Two choices:

1. Split the batch into two copies
   - needs PCIe copy, sync!

2. Keep the batch, skip some packets on GPUs
Predicated execution in a batch

PacketBatch

GPUElement-1

Predicate 0  Predicate 1

GPUElement-2

GPUElement-3

.
.
.

Debatcher

All Packets

Dispatcher

Predicate 0  Predicate 1
Predicated execution in a batch

void ge2kernel(...) {
  if (pkt.predicates[0]) {
    ... // GPUElement-2 logic
  }
}
Predicated execution in a batch

void ge2kernel(...) {
  if (pkt.predicates[0]) {
    ... // GPUElement-2 logic
  }
}

void ge3kernel(...) {
  if (pkt.predicates[1]) {
    ... // GPUElement-3 logic
  }
}
Predicated execution in a batch

```c
void ge2kernel(...) {
    if (pkt.predicates[0]) {
        ...
        // GPUElement-2 logic
    }
}

void ge3kernel(...) {
    if (pkt.predicates[1]) {
        ...
        // GPUElement-3 logic
    }
}

if (pkt.predicates[0])
    output(0).push(pkt);
else if (pkt.predicates[1])
    output(1).push(pkt);
...
```
Apply generic principles

- Batched processing
- Asynchronous non-blocking GPU programming
- Reduce PCIe data transfer with compacted workload
- All use the same locked memory for zero-copy DMA
Apply generic principles

- Asynchronous non-blocking GPU programming

- Reduce PCIe data transfer with compacted workload
- All use the same locked memory for zero-copy DMA
Apply generic principles

- Asynchronous non-blocking GPU programming
- Reduce PCIe data transfer with compacted workload
- All use the same locked memory for zero-copy DMA
Apply generic principles

- Asynchronous non-blocking GPU programming
  - Each batch binds a CUDA stream

- Reduce PCIe data transfer with compacted workload
- All use the same locked memory for zero-copy DMA
Apply generic principles

- Asynchronous non-blocking GPU programming
  - Each batch binds a CUDA stream
  - BEElement only does async GPU ops

- Reduce PCIe data transfer with compacted workload
  - All use the same locked memory for zero-copy DMA
Apply generic principles

- Asynchronous non-blocking GPU programming
  - Each batch binds a CUDA stream
  - BEElement only does async GPU ops
  - Use GPUCompletionQueue to check stream status

- Reduce PCIe data transfer with compacted workload
- All use the same locked memory for zero-copy DMA
Apply generic principles

- Reduce PCIe data transfer with compacted workload
- All use the same locked memory for zero-copy DMA
Apply generic principles

- Reduce PCIe data transfer with compacted workload
- “Region-of-interest” (ROI) based packet slicing
- All use the same locked memory for zero-copy DMA
Packet slicing for ``use-partial''

Packet Classification’s regions-of-interest (ROI):

Packet
Packet slicing for ``use-partial"

Packet Classification’s regions-of-interest (ROI):

```
Protocol

Packet
```
Packet slicing for ``use-partial’’

Packet Classification’s regions-of-interest (ROI):

Packet

Protocol

Src IP
Packet slicing for `use-partial`”

Packet Classification’s regions-of-interest (ROI):

Packet

Src IP

Dst IP

Protocol

Packet
Packet slicing for `use-partial`”

Packet Classification’s regions-of-interest (ROI):

```
Packet
```

```
<table>
<thead>
<tr>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src IP</td>
</tr>
<tr>
<td>Dst IP</td>
</tr>
<tr>
<td>Src Port</td>
</tr>
</tbody>
</table>
```

Packet
Packet slicing for ``use-partial”

Packet Classification’s regions-of-interest (ROI):

Packet

Protocol
Src IP
Dst IP
Src Port
Dst Port
Packet slicing for `use-partial`

Packet Classification’s regions-of-interest (ROI):

Packet

<table>
<thead>
<tr>
<th>ROIs</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src IP</td>
<td>Dst IP</td>
</tr>
<tr>
<td>Src Port</td>
<td>Dst Port</td>
</tr>
</tbody>
</table>
Packet slicing for `use-partial`

Packet Classification’s regions-of-interest (ROI):

Packet slicing

Src IP  Dst IP  Src Port  Dst Port

Protocol

Packet

ROIs
Packet slicing for ``use-partial''

Packet Classification’s regions-of-interest (ROI):

Packet

Slicing

Coalescable memory access on GPUs
Apply generic principles

- Batched processing
- Asynchronous non-blocking GPU programming
- Reduce PCIe data transfer with compacted workload
- All use the same locked memory for zero-copy DMA
Apply generic principles

- All use the same locked memory for zero-copy DMA
Apply generic principles

- All use the same locked memory for zero-copy DMA
- ROIs stored in CUDA locked memory
Evaluation

• Basic forwarding I/O improvement
• Example applications:
  • Classification
  • IP routing
  • Pattern matching for IDS
• Latency, re-order
• Flexibility and modularity
Classifier+routing+pattern matching

![Diagram showing throughput (Gbps) vs. packet size (Bytes) for different applications: Click, Snap-CPU, Snap-GPU, and Snap-GPU w/ Slicing.](c) IDS Router
Classifier+routing+pattern matching

Figure 5.8. Performance of Click and Snap with three different applications.

(c) IDS Router
Evaluation
Evaluation

(a) SDN Forwarder

(b) DPI Router

(c) IDS Router

Figure 5.8

Throughput (Gbps)

Packet Size (Bytes)

64 128 256 512 1024 1518

10 20 30 40

Throughput (Gbps)

Packet Size (Bytes)

64 128 256 512 1024 1518

10 20 30 40

Throughput (Gbps)

Packet Size (Bytes)

64 128 256 512 1024 1518

10 20 30 40

Throughput (Gbps)

Packet Size (Bytes)

64 128 256 512 1024 1518

10 20 30 40

Throughput (Gbps)

Packet Size (Bytes)

64 128 256 512 1024 1518

10 20 30 40
Evaluation

- GPU reached 40Gb/s line rate at 128B
- CPU just 1/3 or 1/4
- Latency tolerable in LAN for non-latency-sensitive app, negligible in WAN
Preserving Click’s flexibility:
full IP router
Preserving Click’s flexibility: full IP router

Add a pattern matching element

Throughput (Gbps)

Packet Size (Bytes)

Click □ Snap-CPU □ Snap-GPU (w/ Slicing)
Preserving Click’s flexibility: full IP router

Add a pattern matching element
Snap Summary

• A generic parallel packet processing framework
  • flexibility of Click
  • fast parallel power from GPUs
  • https://github.com/wbsun/snap
• Details not presented: network I/O, async scheduling
The throughput of system software with parallelizable, computationally expensive tasks can be improved by using GPUs and frameworks with memory-efficient and throughput-oriented designs.
Conclusion
Conclusion

Generic principles:

- Batched processing
- Asynchronous non-blocking GPU programming
- Reduce PCIe data transfer with compacted workload
- All use the same locked memory for zero-copy DMA
Conclusion

**Generic principles:**
- Batched processing
- Asynchronous non-blocking GPU programming
- Reduce PCIe data transfer with compacted workload
- All use the same locked memory for zero-copy DMA

**Concrete frameworks:**
- GPUstore with high throughput storage applications
- Snap with high throughput network packet processing
Thanks!

Q&A
Backup slides
ROI for memory access coalescing
How ROI slicing works?

• To Click insiders:
  • Batcher accepts ROI requests from BElements
  • Batcher merges requested ROIs into result ROIs
  • Each BElement asks for its ROIs’ offsets
  • GPU kernels invoked by BElements use variables for offsets
How predicated execution works?

- To Click insiders:
  - Manually assigned: where and what!
Path-encoded predicate