ActivePointers: The case for software address translation on GPUs

Sagi Shahar
Shai Bergman
Mark Silberstein

Technion – Israel Institute of Technology
History of the world in 7 slides
CPU
ActivePointers  Shahar, Bergman, Silberstein - EE, Technion
CPU
Application

OS

CPU

GPU
Application

OS

CPU

GPU
This is our world
Application

OS services
(GPUfs, GPUnet, GPUrdma)

GPU
Agenda

- Motivation
- Background – GPU file system support
- ActivePointers: software translation layer for `mmap`
- Evaluation
- Conclusions
Motivation:
Processing large datasets on GPUs

Image collage: for every block find “the best match” image in a DB
Image collage: data driven access

For each input image block

- Compute indexes into DB file
- Read candidates from DB file
- Brute-force search to choose the best

Common pattern for DBs, text and image search
Traditional programming model

**GPU as a co-processor**

For each input image block

- Compute **indexes** into DB file
- Read candidates from DB file
- Brute-force search to choose the best
For each input image block

- Compute **indexes** into DB file
- Read candidates from DB file
- Brute-force search to choose the best
File-system access from GPU

Peer-processor model

For each input image block
- Compute **indexes** into DB file
- Read candidates from DB file
- Brute-force search to choose the best

**Easier programming**
**Higher performance**

**GPUfs**
[ASPLOS13]
GPUfs: FS API + Distributed Buffer Cache

Silberstein et al., ASPLOS 2013, TOCS 2014, CACM 2014
Wanted: `mmap()`

- Typical usage: `mmap` the whole file into application address space
- File access through a regular pointer

Benefits

- Simplicity
- On-demand data transfer
- Performance
Wanted: `mmap()`

- Typical usage: `mmap` the whole file into application address space
- File

  Challenge: lack of GPU hardware support for VM management

  Simplicity
  On-demand data transfer
  Performance
Reminder: `mmap` on CPU

- allocate virtual memory region
  - no physical memory allocated first
- on first access - page fault
  - allocate page in a buffer cache
  - read from file
  - map the page into process's virtual address space
Reminder: \texttt{mmap} on CPU

- allocate virtual memory region
  - no physical memory allocated first
- on first access - page fault
  - allocate page in a buffer cache
  - read from file
  - map the page into process's virtual address space

Can we implement \texttt{mmap} on GPU?
mmap on GPU?

- allocate virtual memory region
  - no physical memory allocated first
- on first access - page fault
  - allocate page in a buffer cache
  - read from file
  - map the page into virtual memory
**mmap** on GPU?

- allocate virtual memory region
  - no physical memory allocated first
- on first access - page fault
  - allocate page in a buffer cache
  - read from file
  - map the page into virtual memory

---

No GPU user control

Recent GPUs

GPUfs

ActivePointers

Shahar,Bergman, Silberstein - EE, Technion
Today: CPU-centric VM management

- **Pros:**
  - compatible with the CPU OS
  - no extra GPU code

- **Cons:**
  - CPU in every page fault
  - CPU-GPU coordination for page cache management
  - No GPU page fault handlers

Co-processor: I/O management and control on the CPU
This work: GPU-centric VM management for I/O operations

- Layered on top of regular VM
- Pros:
  - no CPU involvement on page faults
  - GPU handles page faults
  - High throughput and low page fault handling latency

Peer-processor

Data, control path and OS I/O abstractions on the GPU
Software Address Translation Layer

- Pros:
  - No costly hardware TLB updates
  - Extensible
  - User-level access
  - Complementary to Hardware VM

Fully compatible with commodity GPUs
Agenda

- Software address translation
- Evaluation
- Conclusions
Desired behavior

- **GPU code**

```c
void* ptr = gmmmap(fd, offset, size)

for (int i = 0; i < size; i++, ptr++)
{
    ptr[threadIdx.x] = 25;
}
```

One page fault on first access

`ptr` must resolve to **buffer cache page**
Desired behavior

- **GPU code**

```c
void* ptr = g_mmap(fd, offset, size)
for (int i = 0; i < size; i++, ptr++)
{
    ptr[threadIdx.x] = 25;
}
```

Requires page table lookup on every access!
Software TLB - inefficient

• One TLB per core (Threadblock) in shared memory

Extra memory accesses for each read
Contention on TLB updates
How to handle TLB invalidations?
ActivePointers
Main design principles

- Minimize page table lookups
  - translation is cached in hardware registers

- Pages locked in the buffer cache as long as they are in use
  - keep reference count for each page
# ActivePtr structure

<table>
<thead>
<tr>
<th>Valid</th>
<th>Buffer Cache Ptr</th>
<th>File offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>NONE</td>
</tr>
</tbody>
</table>

64 bits
ActivePtr ptr;
ptr = gmmap(fd, 4096, size);
float x = *ptr;
ptr++;
float y = *ptr;
ptr+=4096;
Example

```c
{  
  ActivePtr ptr;
  ptr = gmmap(fd, 4096, size);
  float x = *ptr;
  ptr++;
  float y = *ptr;
  ptr+=4096;
}
```
Example

```
ActivePtr ptr;
ptr = gmmap(fd, 4096, size);
float x = *ptr;
ptr++;
float y = *ptr;
ptr+=4096;
```
Example

Page Table Entry

<table>
<thead>
<tr>
<th>RefCount</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4096</td>
</tr>
</tbody>
</table>

Valid Buffer Cache Ptr File offset

| ptr | 0 | NONE | 4096 |

Pagefault handler

```c
ActivePtr ptr;
ptr = gmmap(fd, 4096, size);
float x = *ptr;
ptr++;
float y = *ptr;
ptr+=4096;
```
Example

```
{ 
  ActivePtr ptr;
  ptr = gmmap(fd,4096,size);
  float x = *ptr;
  ptr++; 
  float y = *ptr;
  ptr+=4096;
}
```
Example

Valid Buffer Cache Ptr File offset

| ptr | 1 | 0xFFF0004 | 4100 |

{ }

ActivePtr ptr;
ptr = gmmap(fd, 4096, size);
float x = *ptr;
ptr++;
float y = *ptr;
ptr+=4096;

ActivePointers Shahar,Bergman, Silberstein - EE, Technion
Example

Page Table Entry

<table>
<thead>
<tr>
<th>RefCount</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4096</td>
</tr>
</tbody>
</table>

0xFFFFC0000

Valid | Buffer Cache Ptr | File offset
--- | ----------------- | -------
ptr  | 1                | 0xFFFFC0004 | 4100

{  
  ActivePtr ptr;
  ptr = gmmap(fd, 4096, size);
  float x = *ptr;
  ptr++;
  float y = *ptr;
  ptr+=4096;
}

Fault-free Lookup-free access
Example

```c
ActivePtr ptr;
ptr = gmmap(fd, 4096, size);
float x = *ptr;
ptr++;
float y = *ptr;
ptr+=4096;
```

Unlock page

Unlock page

Crossing page boundary

About Image:

- **ActivePointers**
- **Shahar, Bergman, Silberstein - EE, Technion**
ActivePtr ptr;
ptr = gmmmap(fd, 4096, size);
float x = *ptr;
ptr++;
float y = *ptr;
// ptr += 4096;

ptr becomes inaccessible
ActivePointers: state machine

uninitialized

Assignment

unlinked

Assignment
Out-of-page arithmetics

Page fault

linked
Challenge:
Thread level address translation

- Reminder: warps = 32 threads in lockstep
- Warp threads may access different pointers
  - Faults for different pages
  - No faults

Divergence! Deadlocks!
Idea:
Translation aggregation mechanism

- Quickly identify fault-free accesses (fast path)
- Handle faults in order
- Aggregate faults to the same page
- Access the page cache using a non-divergent control flow
Not covered in this talk...

- Translation aggregation algorithm
- Integration with GPUfs [SYSTOR16]
  - Highly concurrent page cache
  - Handling 4K pages
- Analysis of software TLB
- Optimizations
Evaluation

- Commodity NVIDIA K80 GPU
- CUDA
- GPUsf
Latency overheads

- **A single** GPU thread performing memory copy using ActivePointers

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Read</th>
<th>Read+Inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw access</td>
<td>225</td>
<td>257</td>
</tr>
<tr>
<td>ActivePointers</td>
<td>271 (+20%)</td>
<td>423 (+65%)</td>
</tr>
</tbody>
</table>
Throughput overheads

- Same experiment with **full GPU occupancy**

<table>
<thead>
<tr>
<th>Transfer bandwidth</th>
<th>4-byte</th>
<th>8-byte</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>99.65 GB/s</td>
<td>148.7 GB/s</td>
</tr>
<tr>
<td></td>
<td>(65.4%)</td>
<td>(97.6%)</td>
</tr>
</tbody>
</table>

Free-computation bubble
Latency hiding – the key to performance

- Different compute/memory ratio

![Graph showing latency sensitivity and hidden latency across different data sizes and threadblock counts.](image-url)
Image collage

- End-to-end evaluation on K80 GPU

  Pointer access to 40GB DB file in GPU memory

- No measurable overhead
- 2.6x over 12 CPU cores with 256-bit AVX
- 3.5x over CPU + GPU
Take aways

- TLB-less address translation:
  - Beyond GPUs? (near-memory computing, FPGAs?)
  - Lightweight hardware support

- **GPU-centric VM management**

- **GPU-as-peer-processor programming model**

Source code at
https://github.com/gpufs
Thank you!

Accelerated Computing Systems Lab

Looking for postdocs

mark@ee.technion.ac.il
Backup
Latency hiding – the key to performance

- Workloads with different compute/IO ratio (float)

![Graph showing latency hiding performance with different workloads and threadblock numbers. The graph compares overhead percentages for various operations such as Add, Read, Random 1, Reduce, Random 10, Bitonic sort, and Random 50, across different data sizes.]
Lowlights

- Compiler heuristics?
Discussion

- Register pressure
- compiler support
- Instructions for boundary checks
- I/O preemption
Major compiler: register pressure

- Active pointer take 2 register (just like standard 64bit pointers)
- Additional meta-data is rarely accessed and can be stored in local memory
- Additional registers are required for page-faults and offset calculations
- Result in reduced optimization opportunities for the rest of the code
Possible ways to cope with it

- Hardware acceleration can replace additional registers in offset calculation
- Most registers are only used in page-fault handling
- These registers are rarely accessed and can be moved to local memory
- The compiler heuristics need to be aware of Active pointers