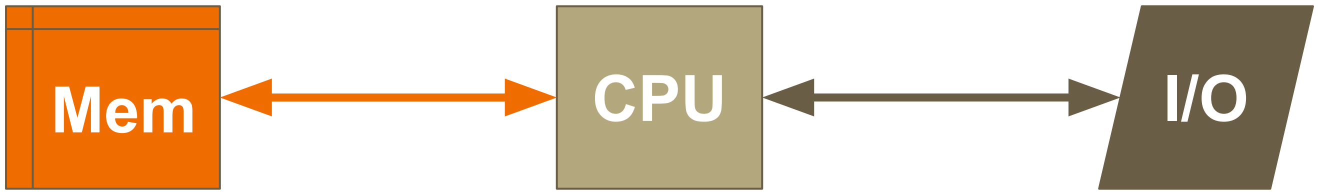
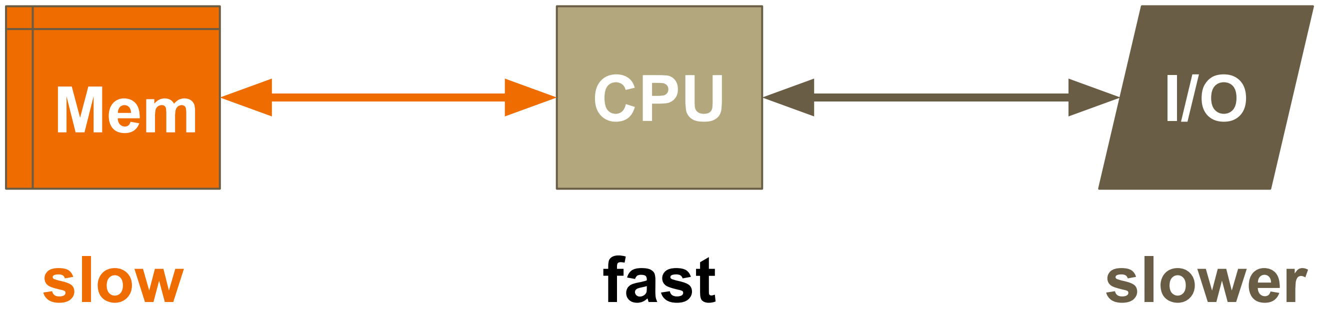
GPUfs: Integrating a File System with GPUs

Yishuai Li & Shreyas Skandan

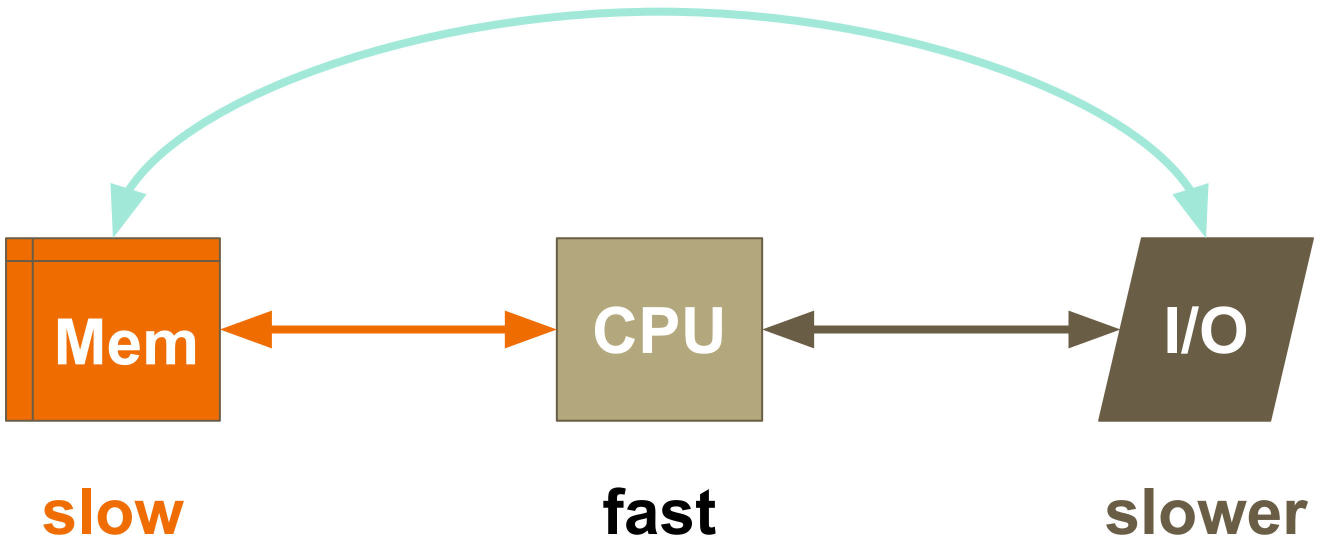
Von Neumann Architecture



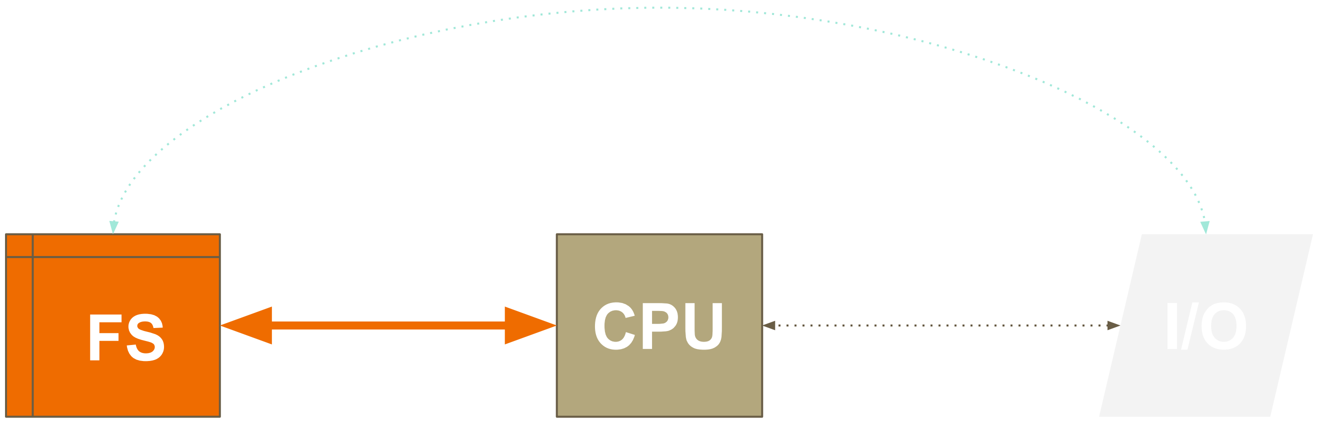
Von Neumann Architecture



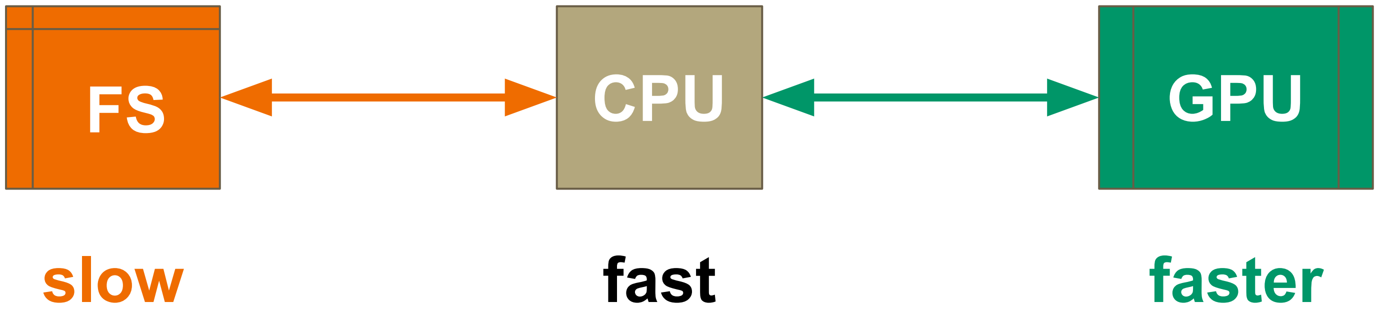
Direct Memory Access



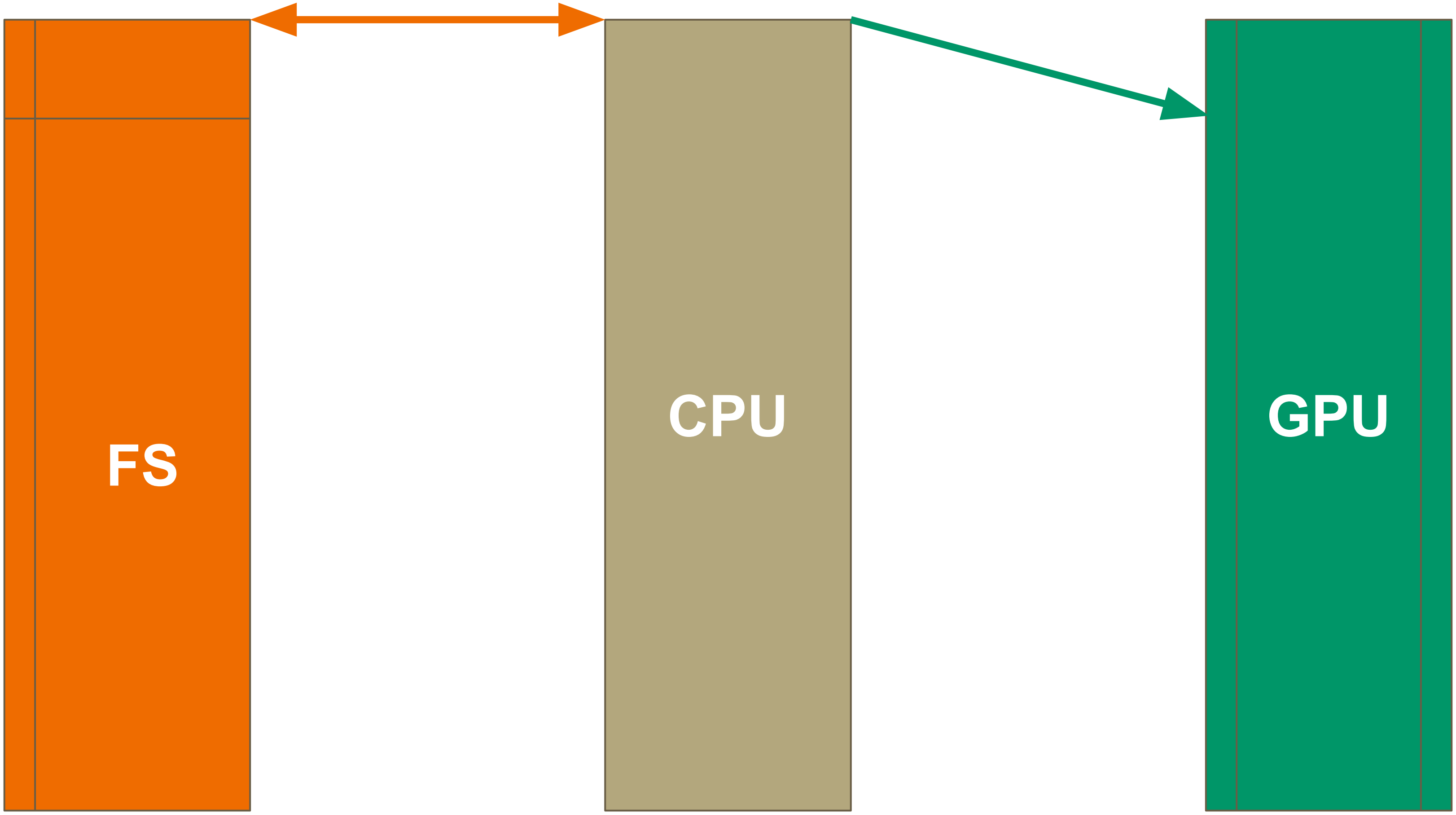
File System



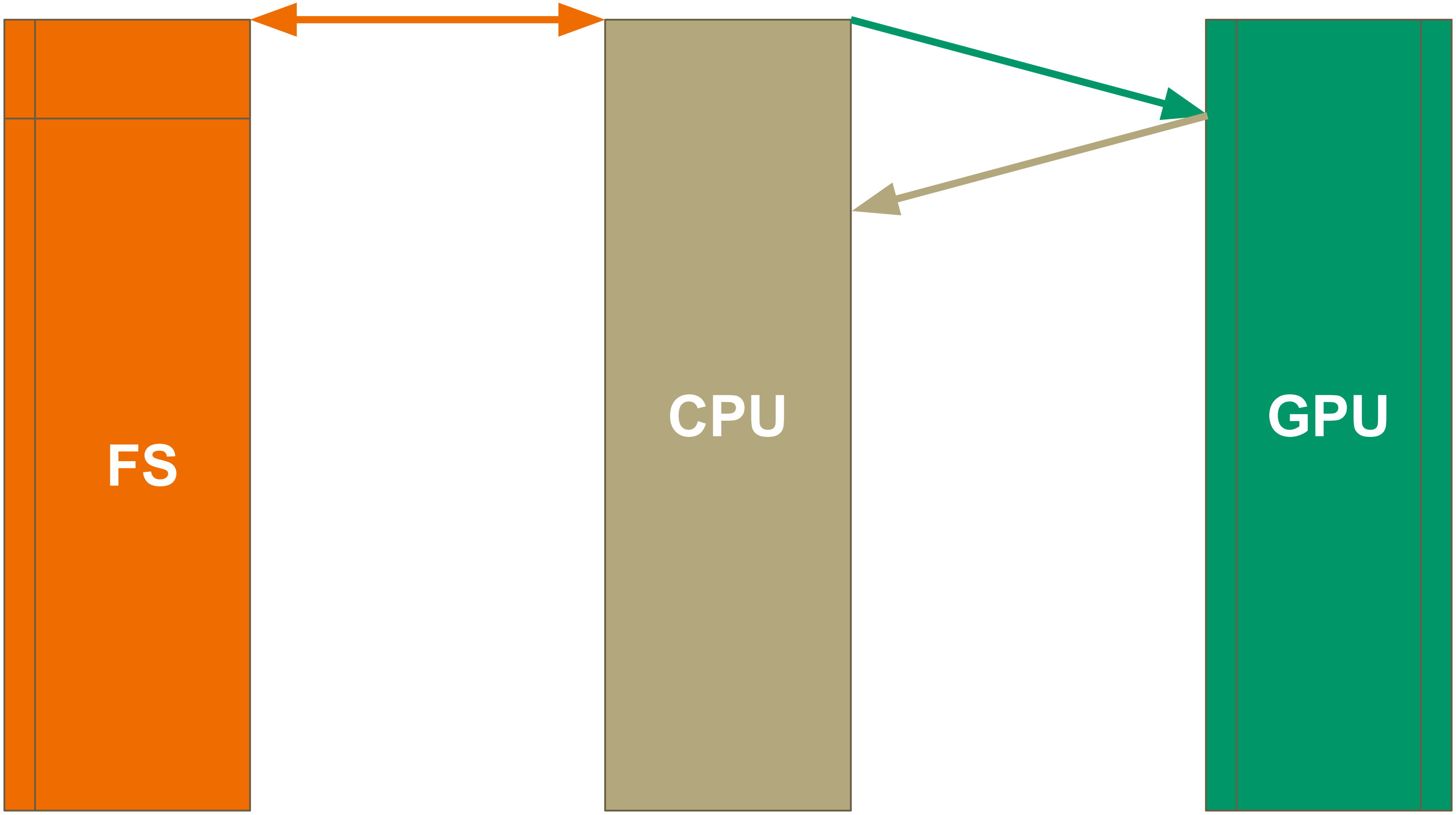
Adding GPU?



Traditional approach

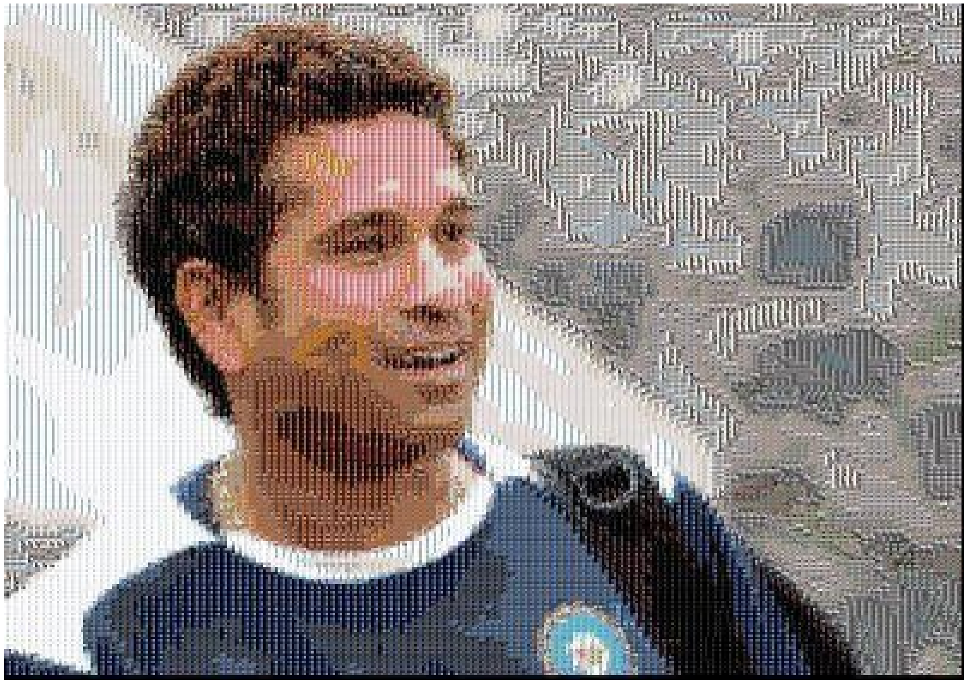


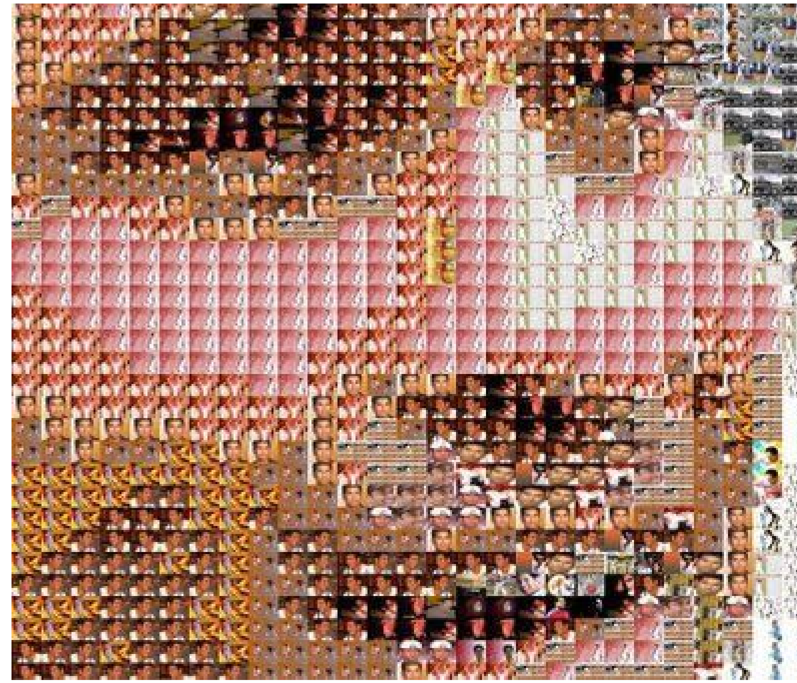
Traditional approach



Face Collage

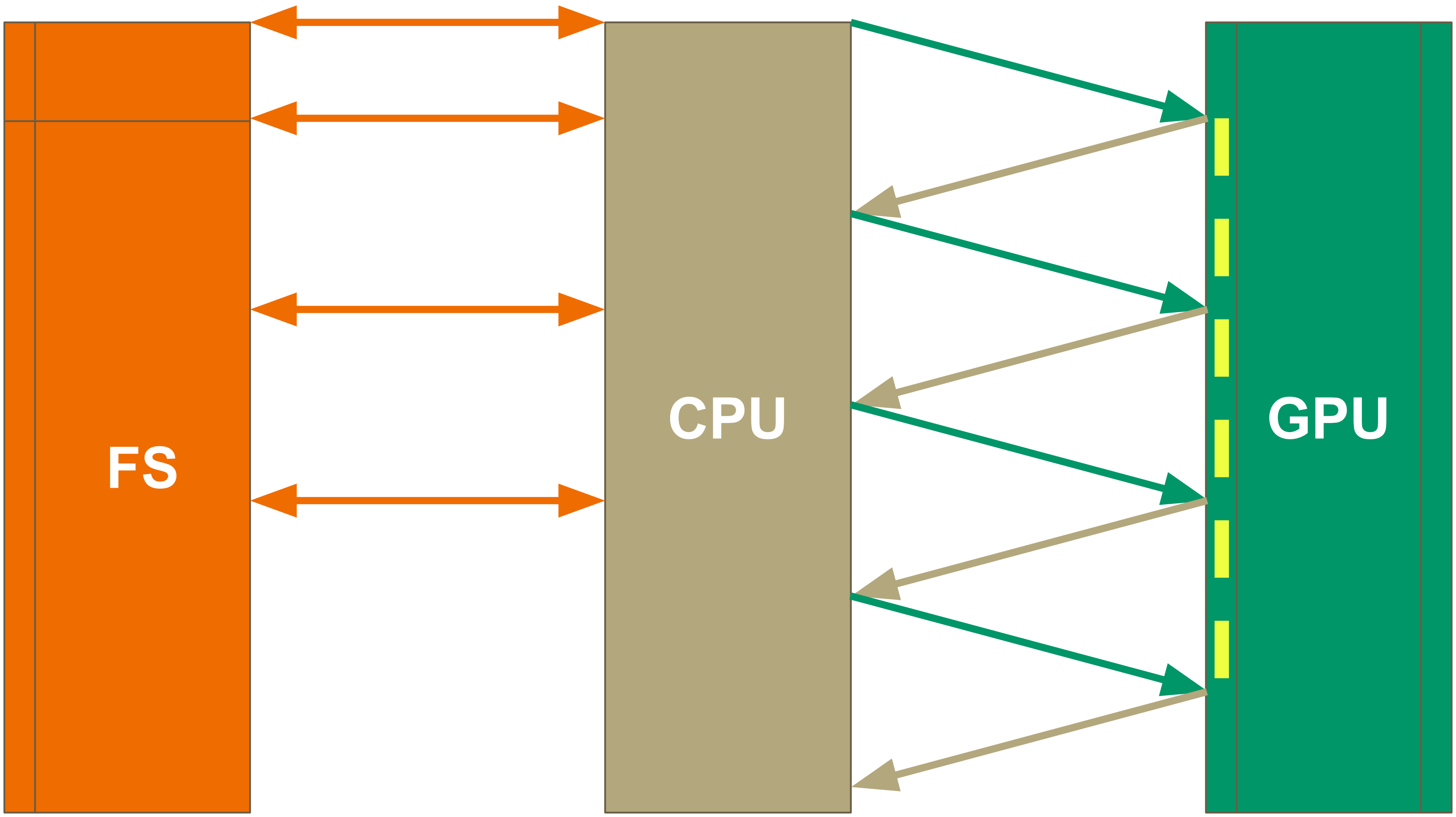
https://www.codeproject.com/articles/36347/face-collage



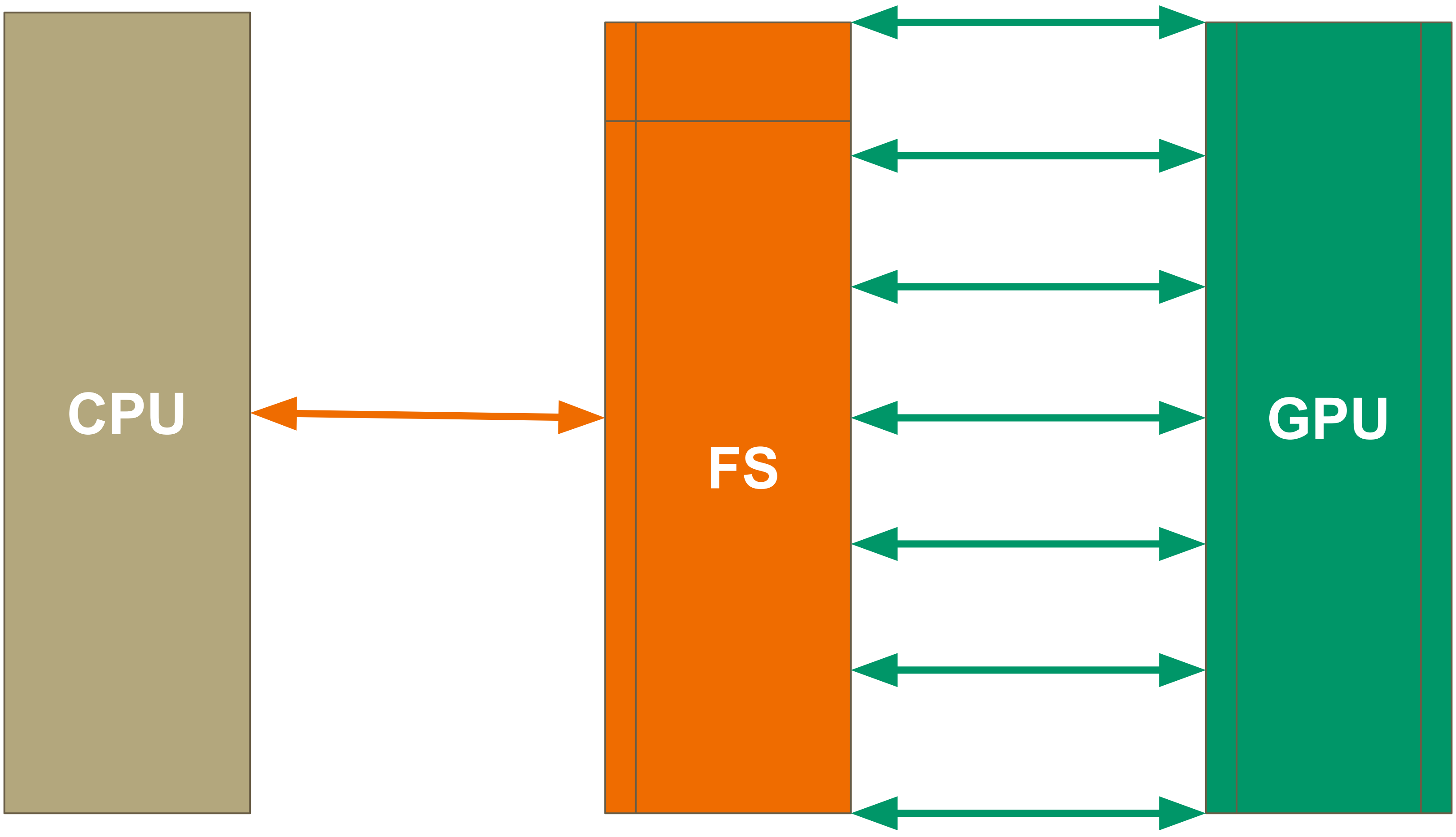


```
 while unhappy do
1.  read next image file
2.  decide placement
3.  remove outliers
 end while
```

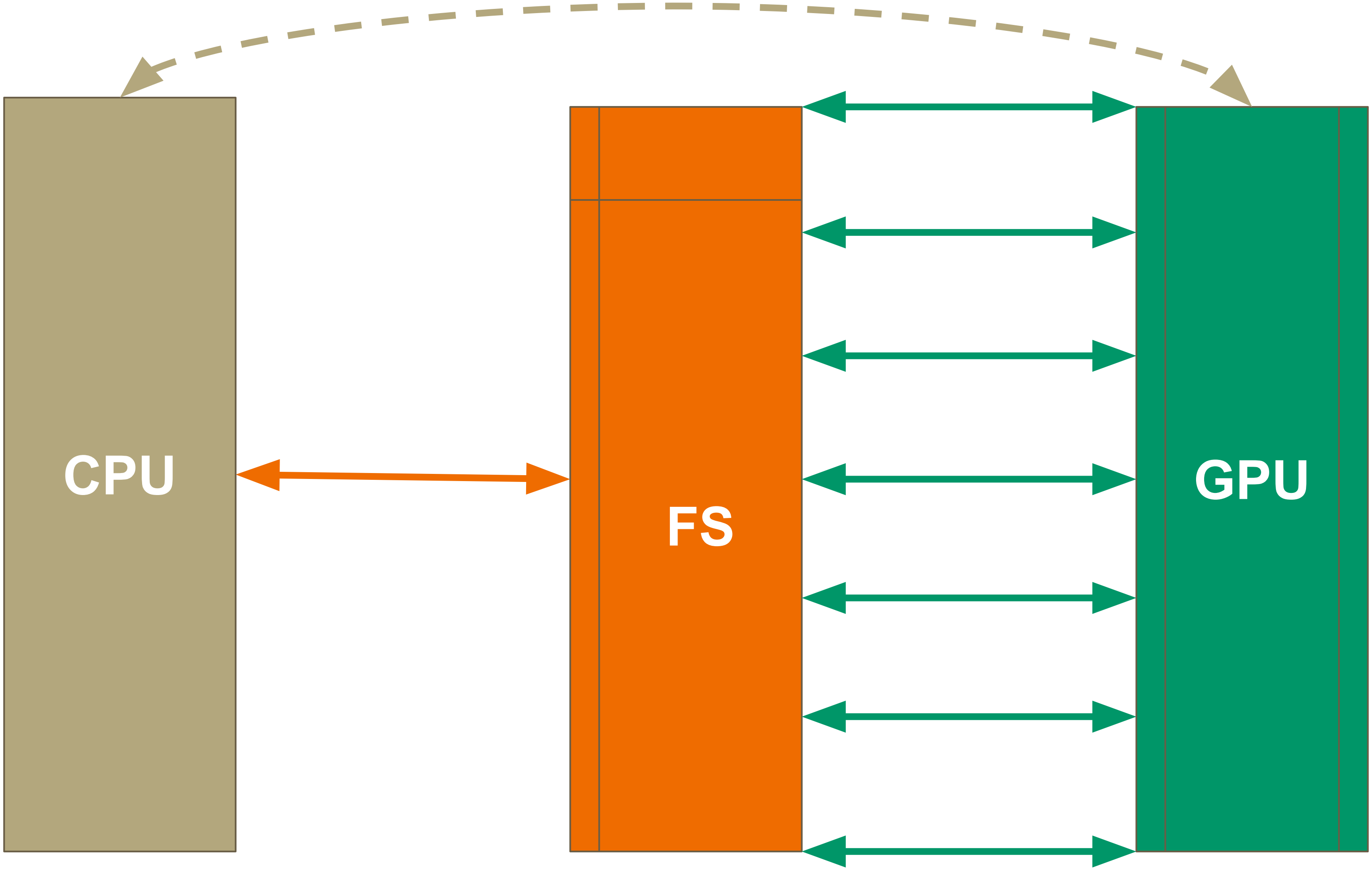
Traditional approach



Ideal structure



Ideal structure



GPU hardware characteristics

Parallelism

Heterogeneous memory

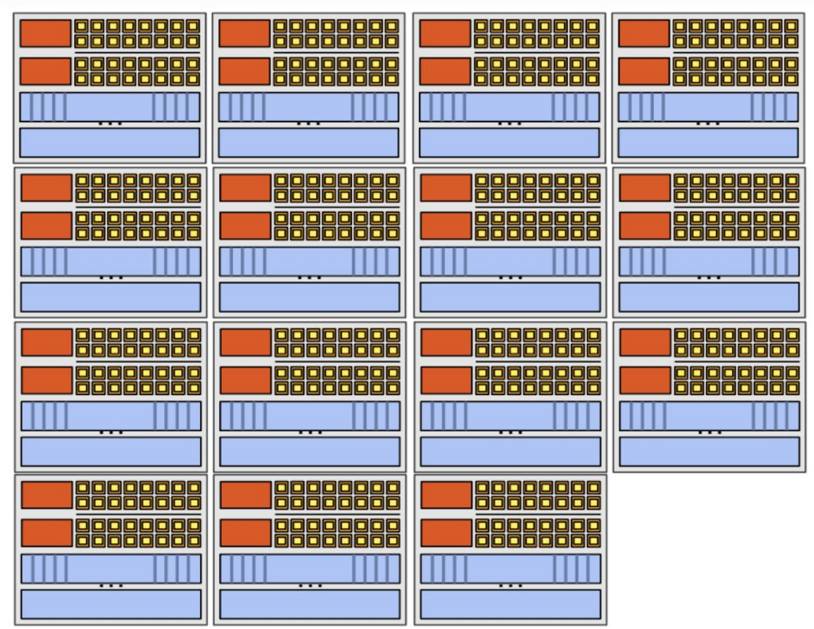
Massive parallelism

NVIDIA Fermi*

23,000
active threads

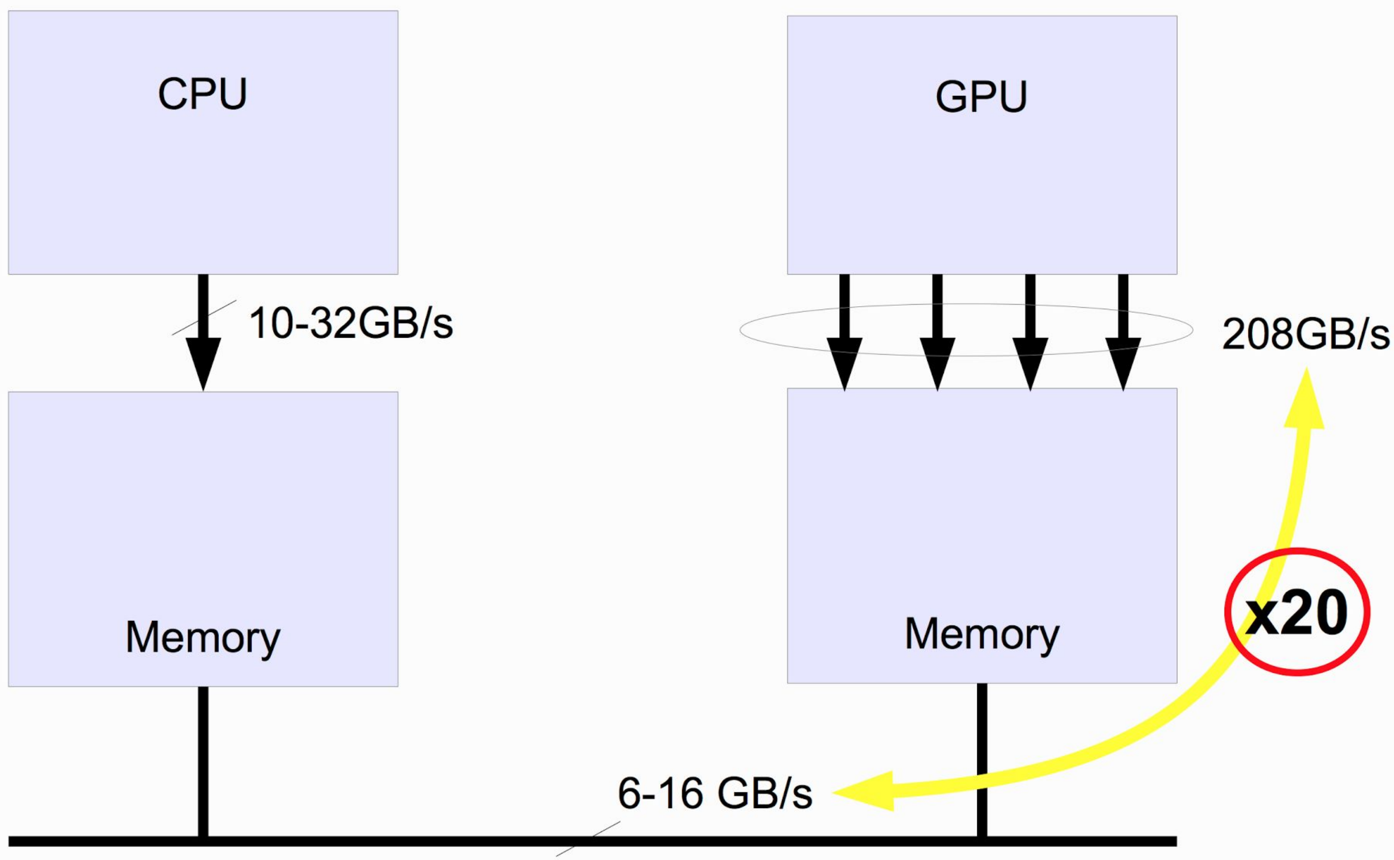
AMD HD5870*

31,000
active threads



From M. Houston/A. Lefohn/K. Fatahalian – A trip through the architecture of modern GPUs*

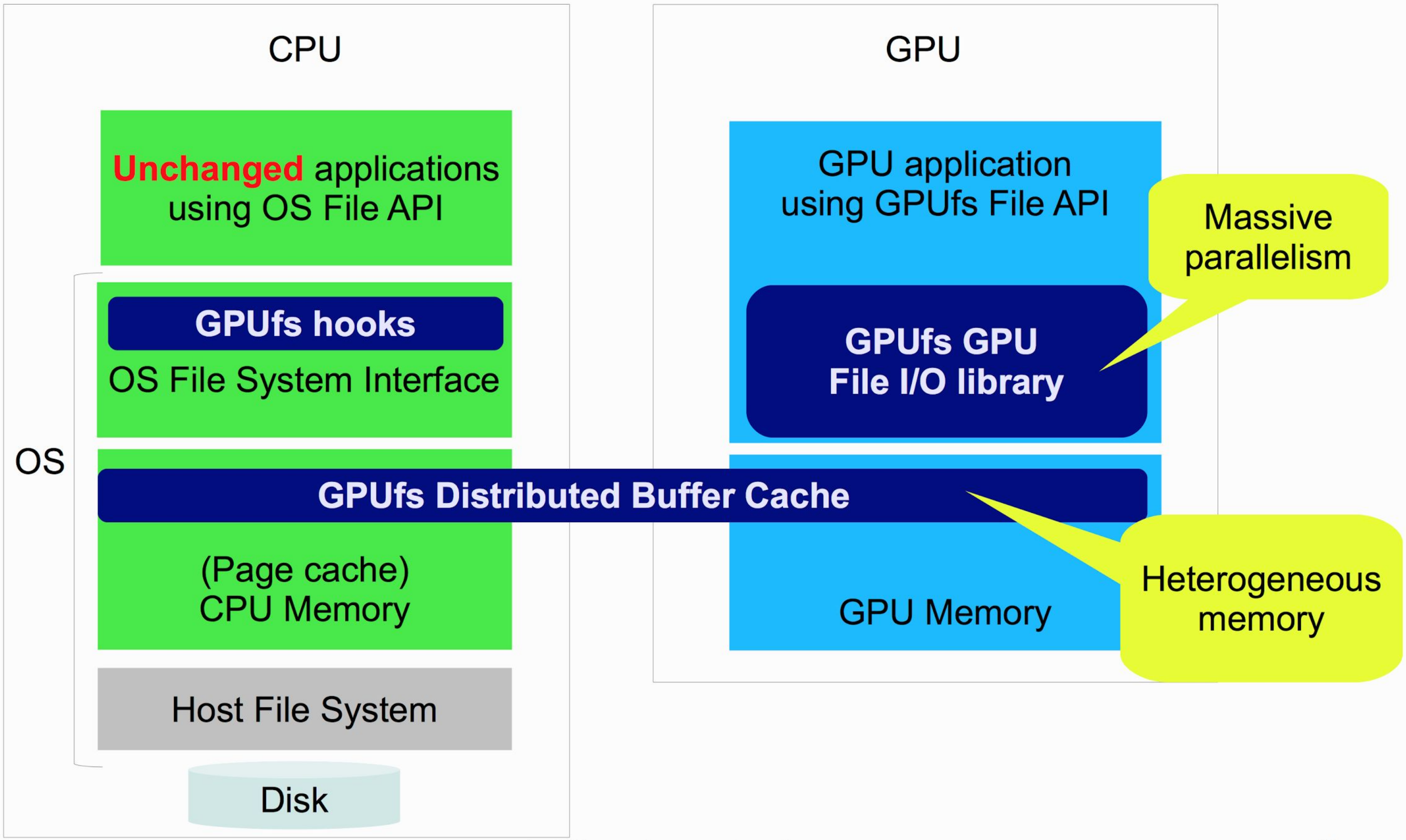
Heterogeneous memory



GPUfs: principled **redesign** of the **whole** file system stack

- **Relaxed FS API semantics** for massive parallelism
- **Relaxed distributed FS consistency** for non-uniform memory
- **GPU-specific implementation** of synchronization primitives, lock-free data structures, memory allocation, ….

High-level design



GPU File I/O API

`open/close` →	`gopen/gclose`
`read/write` →	`gread/gwrite`
`mmap/munmap` →	`gmmap/gmunmap`
`fsync/msync` →	`gfsync/gmsync`
`ftrunc` →	`gftrunc`

Should we preserve CPU semantics?

Parallel square root on CPU

```
cpu_thread(thread_id i){
 float buffer;
 int fd=open(filename,O_GRDWR);

 offset=sizeof(float)*i;
 pread(fd,sizeof(float),&buffer,offset);
 buffer=sqrt(buffer);
 pwrite(fd,sizeof(float),&buffer,offset);

 close(fd);
}
```

Thread i

find i-th element in file

read a_i

$a_i \leftarrow \sqrt{a_i}$

write new a_i

Parallel square root on GPU

```
gpu_thread(thread_id i){
 float buffer;
 int fd=gopen(filename,O_GRDWR);

 offset=sizeof(float)*i;
 gread(fd,sizeof(float),&buffer,offset);
 buffer=sqrt(buffer);
 gwrite(fd,sizeof(float),&buffer,offset);

 gclose(fd);
}
```

This code runs in 100,000 **GPU** threads

Structured parallelism problem

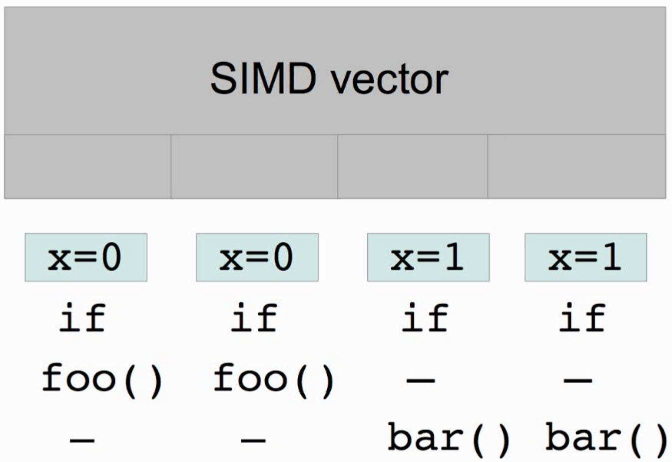
```
float buffer;
```

SIMD divergence

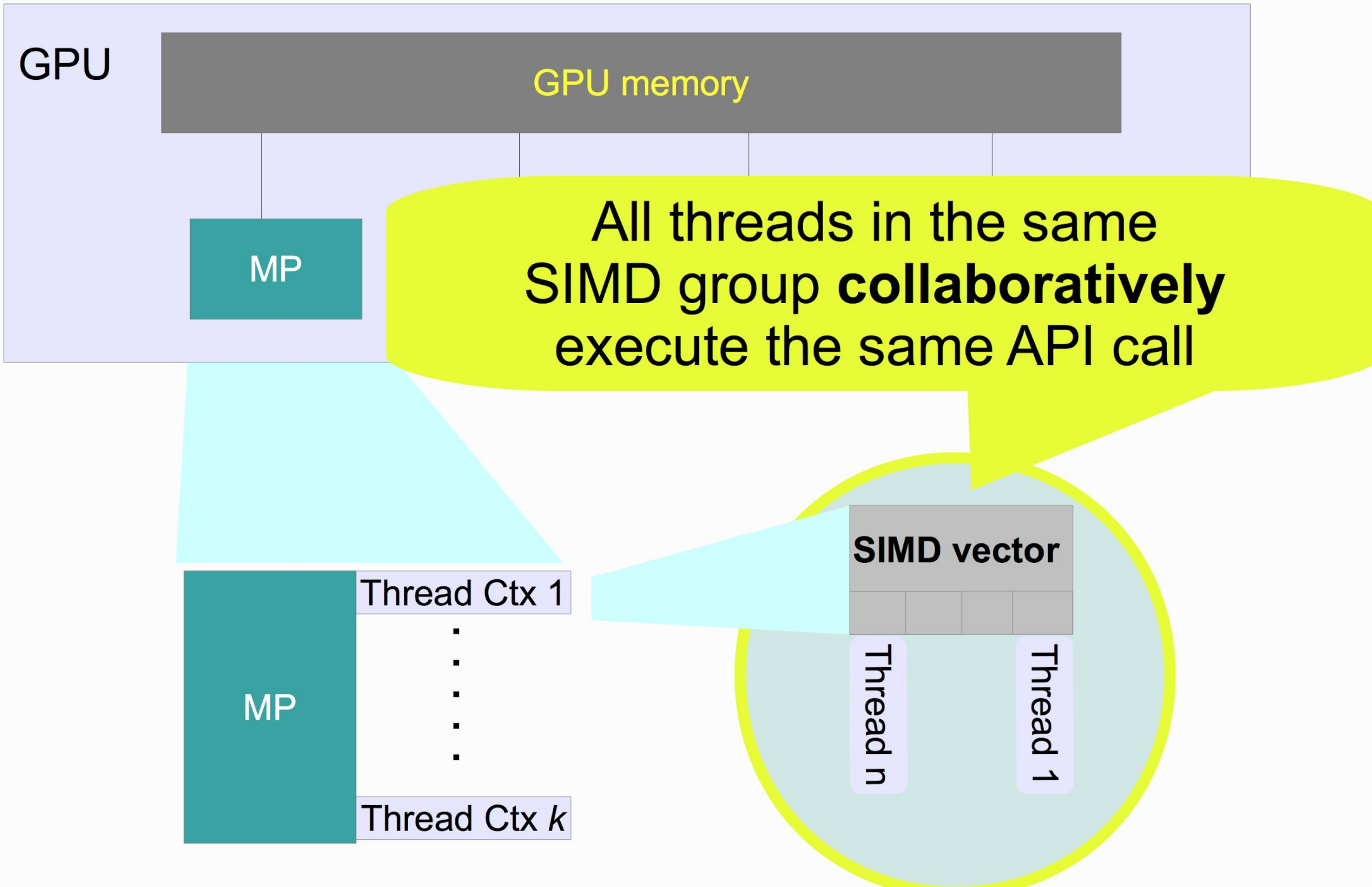
```
int fd=gopen(filename,O_GRDWR);
```

```
if(x==0){
   foo();
}
else{
   bar();
}
```

Reminder:



API call granularity



Too many opens

```
sqrt_gpu(char* filename ){

int fd=gopen(filename,O_GRDWR);
```

Do we need many FDs?

Only 1 real open on CPU

```
sqrt_gpu(char* filename ){

  int fd=gopen(filename,O_GRDWR);
```

gopen() is cached
on GPU
1 FD per file

Parallel read/write

```
sqrt_gpu(char* filename ){

 int fd=gopen(filename,O_GRDWR);
 offset=BLOCK_SIZE*sizeof(float)*blockIdx.x;
 gread(fd,offset,&buffer,BLOCK_SIZE*sizeof(float));

 buffer[threadIdx.x]=sqrt(buffer[threadIdx.x]);

 gwrite(fd,offset,&buffer,BLOCK_SIZE*sizeof(float));
```

When to sync?

```
sqrt_gpu(char* filename ){

 int fd=gopen(filename,O_GRDWR);
 offset=BLOCK_SIZE*sizeof(float)*blockIdx.x;
 gread(fd,offset,&buffer,BLOCK_SIZE*sizeof(float));

 buffer[threadIdx.x]=sqrt(buffer[threadIdx.x]);

 gwrite(fd,offset,&buffer,BLOCK_SIZE*sizeof(float));

 gclose(fd);
}
```

When to sync?

GPU: `gclose()` will eventually sync?

Option 1: Let GPU sync asynchronously

NO – No GPU threads

Option 2: Let CPU sync asynchronously

NO – GPU-CPU atomics necessary

Sync on *last* close?

- **No: hardware non-deterministic scheduling**

Which call is the last one?

```
fd=gopen("file");
gclose(fd);
```

- Kernel invoked in 3 threads

Sync on *last* close?

- **No: hardware non-deterministic scheduling**

Which call is the last one?

```
fd=gopen("file");
gclose(fd);
```

- Kernel invoked in 3 threads
- Run 1: o,o,o,c,c,c

sync here

Run 2: o,o,c,c,o,c

spurious sync here

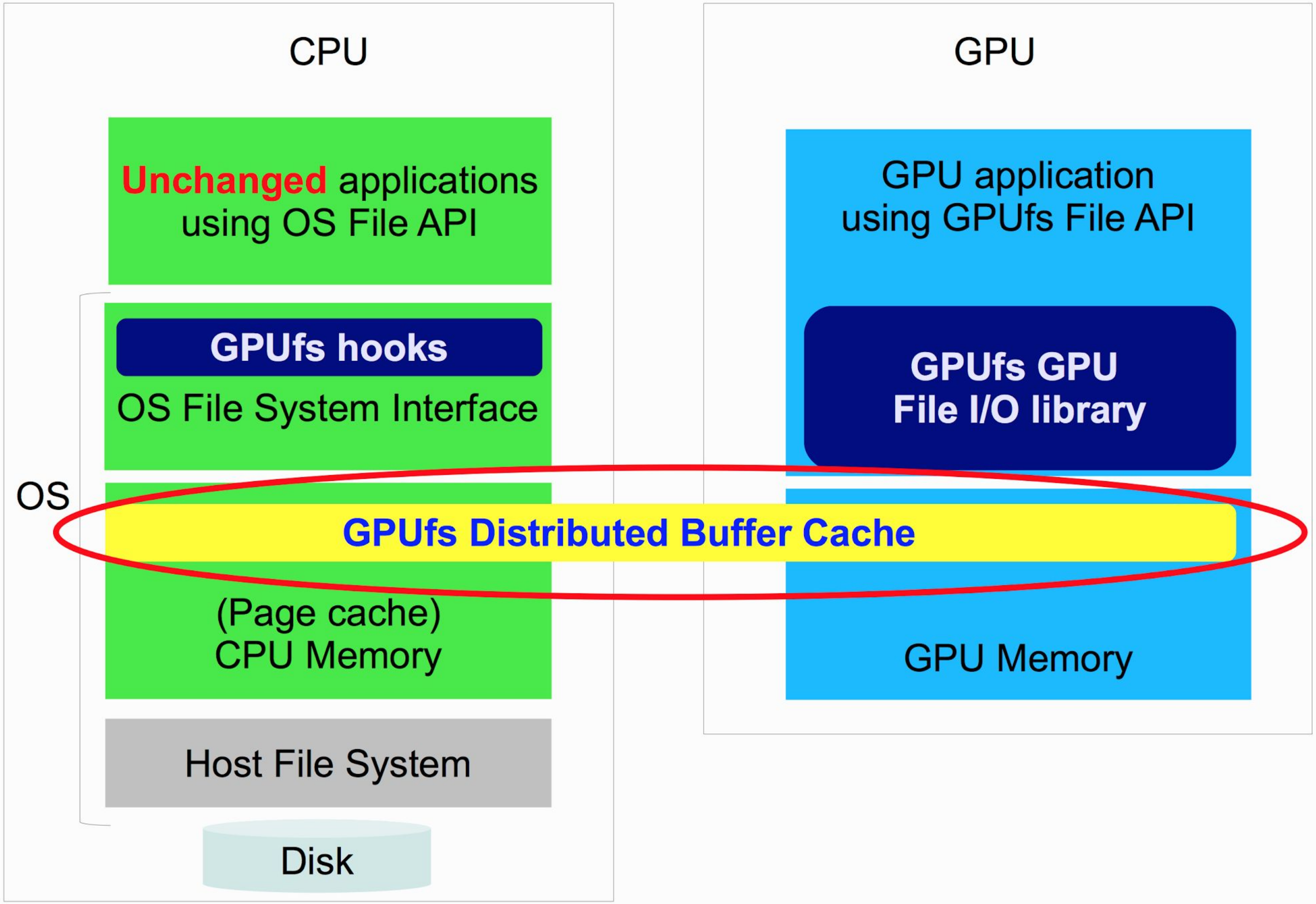
sync here

GPUfs API semantics

```
sqrt_gpu(char* filename ){

 int fd=gopen(filename,O_GRDWR);
 offset=BLOCK_SIZE*sizeof(float)*blockIdx.x;
 gread(fd,offset,&buffer,BLOCK_SIZE*sizeof(float));

 buffer[threadIdx.x]=sqrt(buffer[threadIdx.x]);

 gwrite(fd,offset,&buffer,BLOCK_SIZE*sizeof(float));
 if (is_last_block()) gfsync(fd);
 gclose(fd);
}
```

sync is decoupled from gclose()

GPUfs high-level design

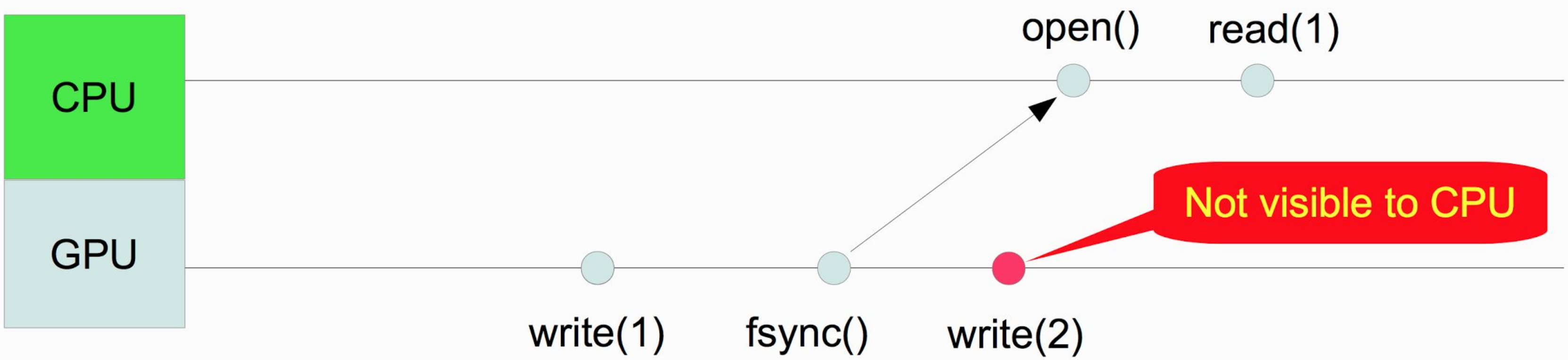


Buffer cache semantics

Local (strong) or
Distributed (weak) file system
data consistency?

Weak data consistency model

- close(sync)-to-open semantics (AFS)



Reason

Minimize inter-processor synchronization

Implications

- Undefined results for overlapping writes
- Explicit sync necessary
- Cache page false sharing

>>

Implementation bits

4. Implementation

GPUfs: Three main software layers

Layer 1 (Core)

- GPUfs API
- Open file states
- Buffer cache

Layer 2 (Communication)

- CPU-GPU comms
- Shared data structures write-shared CPU memory
- CPU-GPU Remote Procedure Call (RPC)

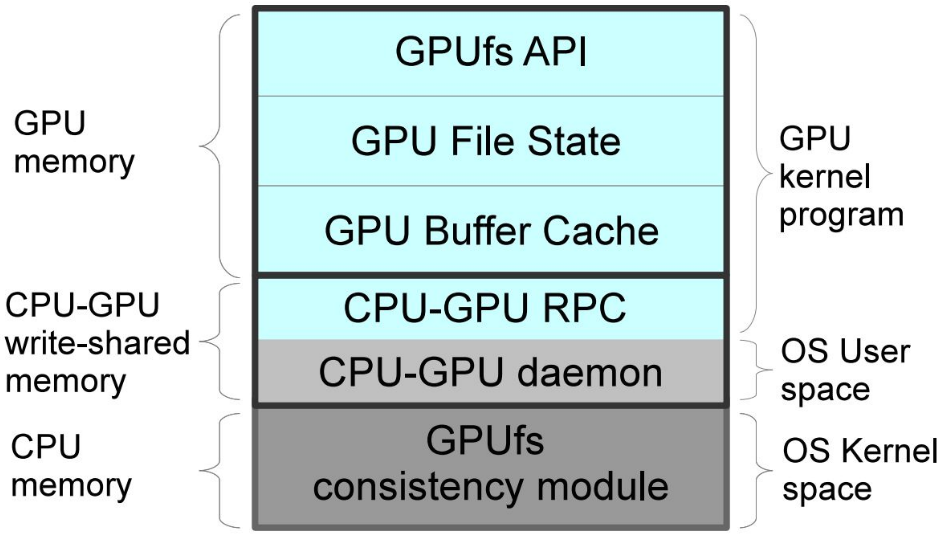


Figure 2. Main GPUfs software layers and their location in the software stack and physical memory.

Layer 3 (Consistence Layer)

- OS Kernel Module
- Consistency between CPU buffer cache and GPU buffer cache

RPC: *"is when a computer program causes a procedure to execute in another address space, which is coded as if it were a normal procedure call, without the programmer explicitly coding it"*

4.1 - File System Operations

OPEN & CLOSE

- Open file table - **pointer** to index of file page, **path**name, cpu **file descriptor**, **reference count** of the number of thread blocks holding the file open
- When closed, **retained in GPU memory**
- Close file table - **pointers** to caches of closed files, **hash table** indexed by **inode**

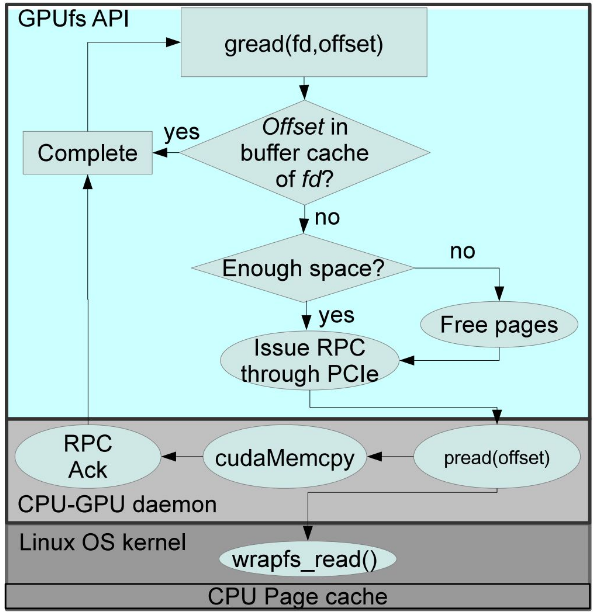


Figure 3. Functional diagram of a call to `gread`. Color scheme is the same as Figure 2.

READ & WRITE

- Check cache for block -> forward request to CPU to allocate cache
- Many threads copy data or initialize pages collaboratively - *"gread"*
- Reference counts protect pages during transfers
- *"gwrites"* end with issuing a memory fence to update GPU memory for consistency

FILE MANAGEMENT

- Generate RPC to the CPU to request the respective operation on the host

4.2 - GPU Buffer Cache

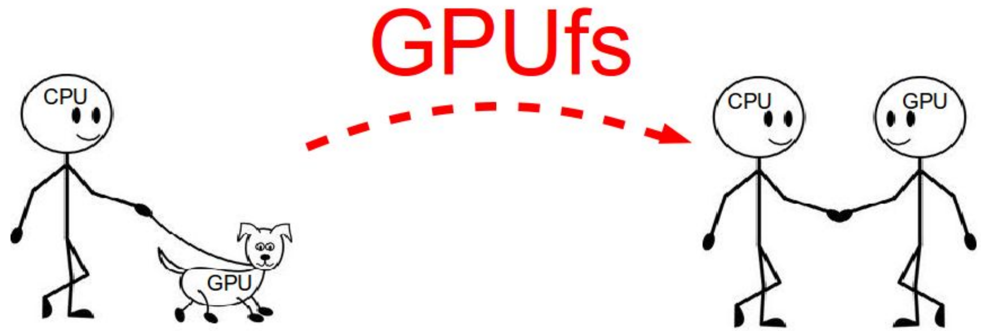
- GPUfs pre-allocates pages - **contiguous memory array** - raw data array
- **PFRAME** structure holds metadata - size, status, offset - allocated in an **array**

- Buffer cache - keeps replicas of previous content - file granularity
- **PAGE LOOKUP** via **dynamic radix tree indexing** of file's buffer cache
  - leaf nodes contain arrays of fpage - containing pframe data
  - Fpage : concurrent access | reference count | spinlock

- **CACHE MANAGEMENT** - Daemon threads are inefficient - own threadblock
- Constantly running or part of each GPU application?
- GPUfs - implements FIFO-like policy for tracking allocation of leaf nodes
  - Newly allocated nodes placed at head of doubly-linked-list
  - To evict a page perform traversal to reclaim a page - closed < read-only

- **CACHE ACCESS** - buffer cache radix tree major point of contention
- To avoid data races - lock free read & locked updates

4.3 - Remote Procedure Calls



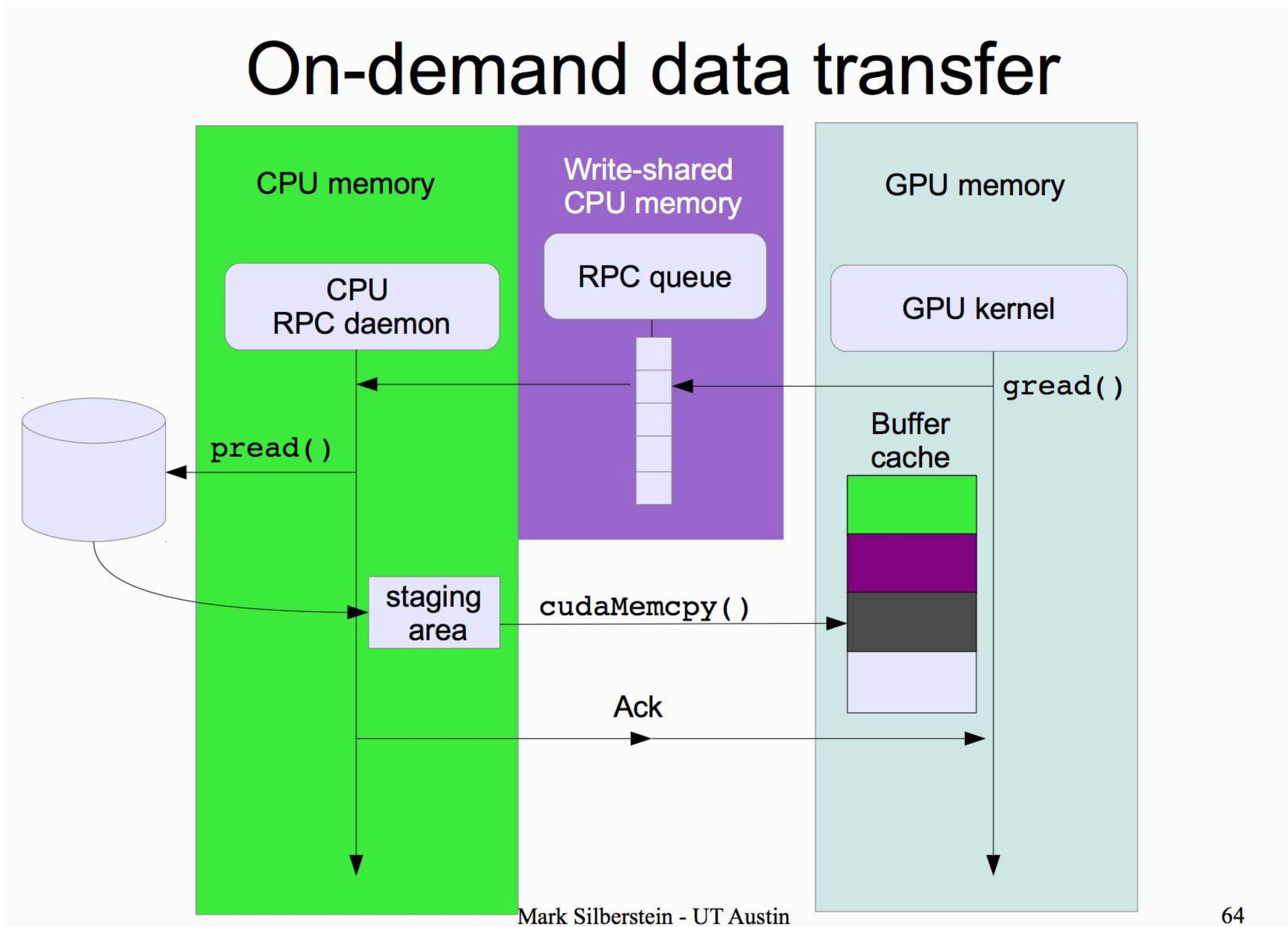
- Co-ordinate data transfer between CPU and GPU
- **PROTOCOL** - Synchronous | client-server protocol | FIFO request channel
- **GPU-AS-CLIENT** instead of GPU-as-coprocessor
- GPU issues request to file server on CPU

- **GPU-CPU MEMORY FENCES**
  - GPU file read and write need to be delivered to CPU when the kernel is running
  - Consistent bi directional updates of the CPU-GPU shared memory

- **GPU CACHE BYPASS**
  - For consistent reads of GPU memory, GPUs must bypass the GPUs L1 and L2 cache to read the CPU initiated memory updates

- Current API - coarse grained notifications when kernel completes. CPU polls the GPU-CPU shared memory region

Implementation Overview



4.4 - File Consistency Management

- **WRITER CONCURRENCY** - only one writer at a time | no diff-and-merge
- Lazy **invalidation propagation** - invalidating the contents of a closed file's cache
- No direct way to push changes on one GPU to another unless reopened

- Uses **WRAPFS** for file consistency - modified for GPUfs
- Software layer over the GPUfs file system - interposition on calls to FS

4.5 - Limitations

- GPU kernels launched by one CPU process cannot access GPU memory of kernels launched by other processes

- GPUfs cannot protect the contents of its GPU buffer caches from corruption by the application it serves

Example Code - 1

```
__global__ void file_cpy_to_gpu(char* src_file)
{

       int zfd=gopen(src,O_GRDONLY);

       int filesize=fstat(zfd);

       for(size_t me=0; me<ONE_BLOCK_READ; me+=FS_BLOCKSIZE)
        {
                int my_offset=blockIdx.x*ONE_BLOCK_READ;

                unsigned int toRead=min((unsigned int)FS_BLOCKSIZE,(unsigned int)
(filesize-me-my_offset));
                volatile void* data=gmmap(NULL, toRead, 0 , O_GRDONLY,zfd,my_offset+me);
        /** ...........................................................
                 process data from file
            ...........................................................
          **/

                gmunmap(data,0);
        }
        gclose(zfd);
}
```

Example Code - 1

```
__global__ void file_cpy_to_gpu(char* src_file)
{
       int zfd=gopen(src,O_GRDONLY);

       int filesize=fstat(zfd);

       for(size_t me=0; me<ONE_BLOCK_READ; me+=FS_BLOCKSIZE)
        {
                int my_offset=blockIdx.x*ONE_BLOCK_READ;

                unsigned int toRead=min((unsigned int)FS_BLOCKSIZE,(unsigned int)
(filesize-me-my_offset));
                volatile void* data=gmmap(NULL, toRead, 0 , O_GRDONLY,zfd,my_offset+me);
        /** ...........................................................
                 process data from file
            ...........................................................
          **/

                gmunmap(data,0);
        }
        gclose(zfd);
}
```

Example Code - 1

```
__global__ void file_cpy_to_gpu(char* src_file)
{

        int zfd=gopen(src,O_GRDONLY);

        int filesize=fstat(zfd);

        for(size_t me=0; me<ONE_BLOCK_READ; me+=FS_BLOCKSIZE)
         {
                int my_offset=blockIdx.x*ONE_BLOCK_READ;

                unsigned int toRead=min((unsigned int)FS_BLOCKSIZE,(unsigned int)
(filesize-me-my_offset));
                volatile void* data=gmmap(NULL, toRead, 0 , O_GRDONLY,zfd,my_offset+me);
         /** ...........................................................
                 process data from file
             ...........................................................
           **/

                gmunmap(data,0);
         }
         gclose(zfd);
}
```

Example Code - 1

```
__global__ void file_cpy_to_gpu(char* src_file)
{

        int zfd=gopen(src,O_GRDONLY);

        int filesize=fstat(zfd);

        for(size_t me=0; me<ONE_BLOCK_READ; me+=FS_BLOCKSIZE)
         {
                int my_offset=blockIdx.x*ONE_BLOCK_READ;

                unsigned int toRead=min((unsigned int)FS_BLOCKSIZE,(unsigned int)
(filesize-me-my_offset));
                volatile void* data=gmmap(NULL, toRead, 0 , O_GRDONLY,zfd,my_offset+me);
         /** .....................................................
                 process data from file
             .....................................................
           **/

                gmunmap(data,0);
         }
         gclose(zfd);
}
```

Example Code - 1

```
__global__ void file_cpy_to_gpu(char* src_file)
{

       int zfd=gopen(src,O_GRDONLY);

       int filesize=fstat(zfd);

       for(size_t me=0; me<ONE_BLOCK_READ; me+=FS_BLOCKSIZE)
        {
                int my_offset=blockIdx.x*ONE_BLOCK_READ;

                unsigned int toRead=min((unsigned int)FS_BLOCKSIZE,(unsigned int)
(filesize-me-my_offset));
                volatile void* data=gmmap(NULL, toRead, 0 , O_GRDONLY,zfd,my_offset+me);
        /** ..........................................................
                 process data from file
            ..........................................................
          **/

                gmunmap(data,0);
        }
        gclose(zfd);
}
```

Example Code - 1

```
__global__ void file_cpy_to_gpu(char* src_file)
{

       int zfd=gopen(src,O_GRDONLY);

       int filesize=fstat(zfd);

       for(size_t me=0; me<ONE_BLOCK_READ; me+=FS_BLOCKSIZE)
        {
               int my_offset=blockIdx.x*ONE_BLOCK_READ;

               unsigned int toRead=min((unsigned int)FS_BLOCKSIZE,(unsigned int)
(filesize-me-my_offset));
               volatile void* data=gmmap(NULL, toRead, 0 , O_GRDONLY,zfd,my_offset+me);
        /** ...........................................................
                process data from file
           ...........................................................
          **/

               gmunmap(data,0);
        }
        gclose(zfd);
}
```

Example Code - 2

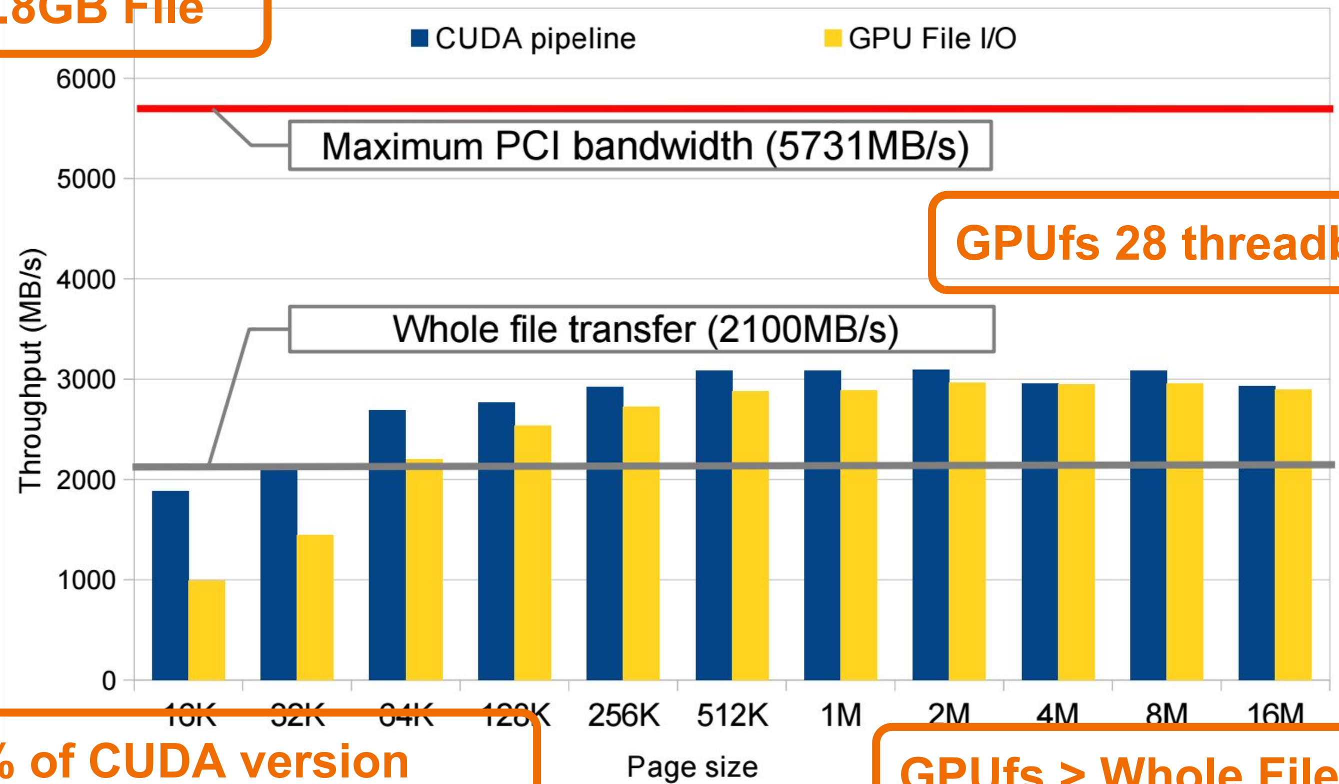
GPUfs API semantics

```
sqrt_gpu(char* filename ){

 int fd=gopen(filename,O_GRDWR);
 offset=BLOCK_SIZE*sizeof(float)*blockIdx.x;
 gread(fd,offset,&buffer,BLOCK_SIZE*sizeof(float));

 buffer[threadIdx.x]=sqrt(buffer[threadIdx.x]);

 gwrite(fd,offset,&buffer,BLOCK_SIZE*sizeof(float));
 if (is_last_block()) gfsync(fd);
 gclose(fd);
}
```

sync is decoupled from gclose()

5. Evaluation - Sequential File Read

Single 1.8GB File



GPUfs 28 threadblocks

Within 5% of CUDA version

GPUfs > Whole File Transfer

Figure 4. Sequential read performance as a function of the page size. The red line is the maximum achievable PCI bandwidth on this hardware configuration. Higher is better.

5. Evaluation - Sequential File Read

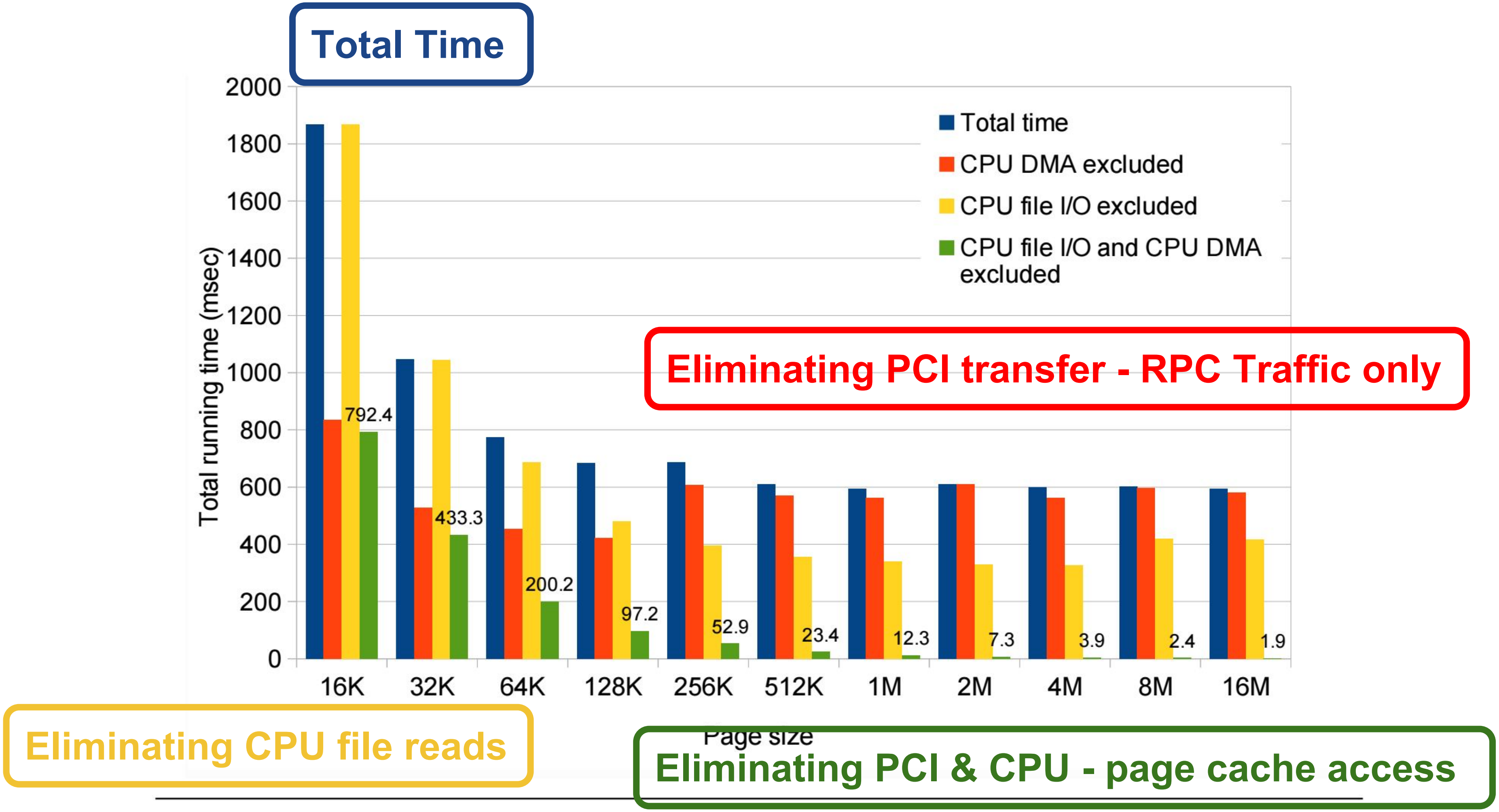


Figure 5. Contribution of different factors to the file I/O performance as a function of the page size. Lower is better.

5. Evaluation - Sequential File Read

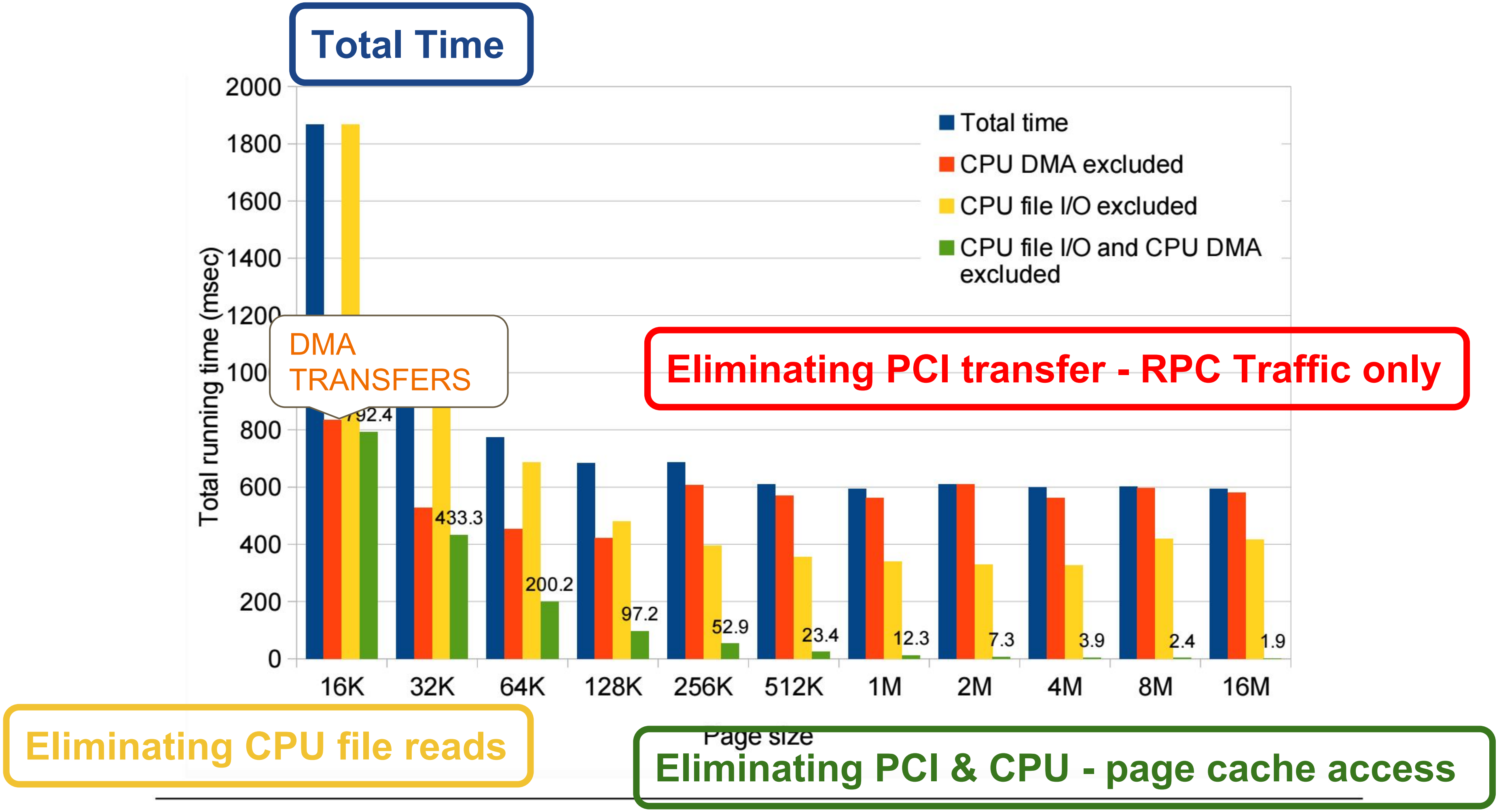


Figure 5. Contribution of different factors to the file I/O performance as a function of the page size. Lower is better.

5. Evaluation - Sequential File Read

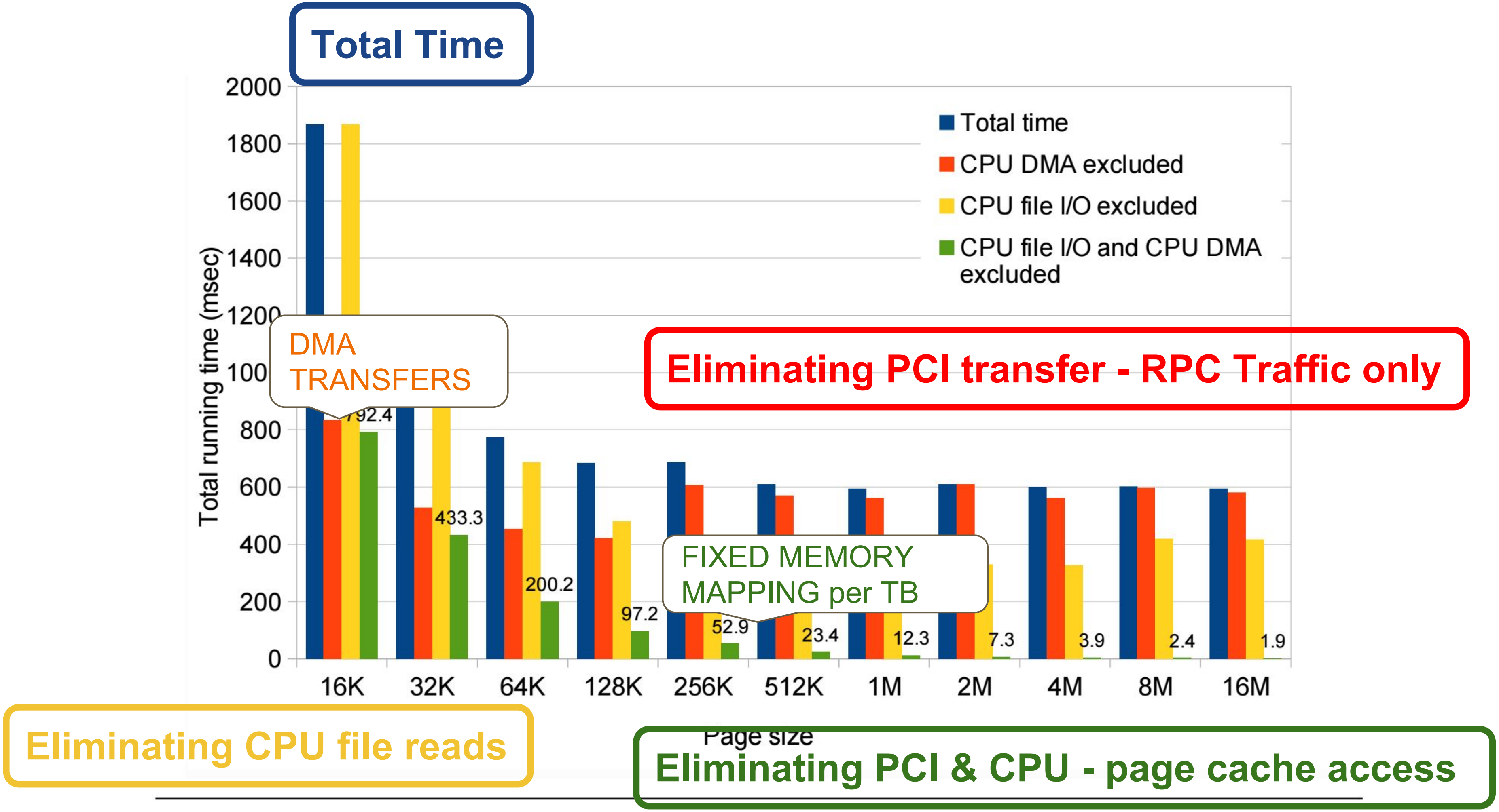
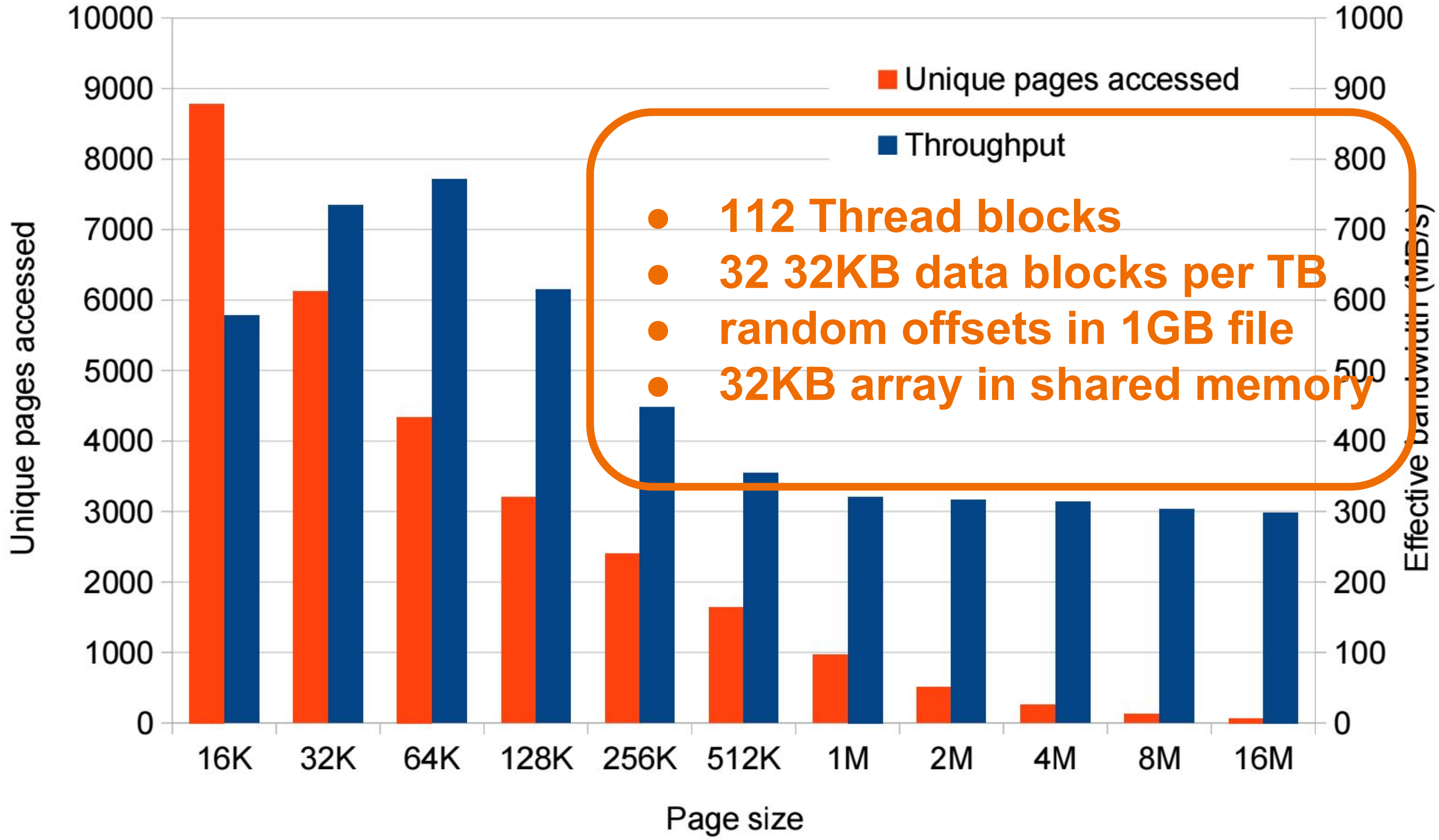


Figure 5. Contribution of different factors to the file I/O performance as a function of the page size. Lower is better.

5. Evaluation - Random File Read



- 112 Thread blocks
- 32 32KB data blocks per TB
- random offsets in 1GB file
- 32KB array in shared memory

Figure 6. Random read/write performance as a function of page size. Higher is better.

5. Evaluation - Random File Read

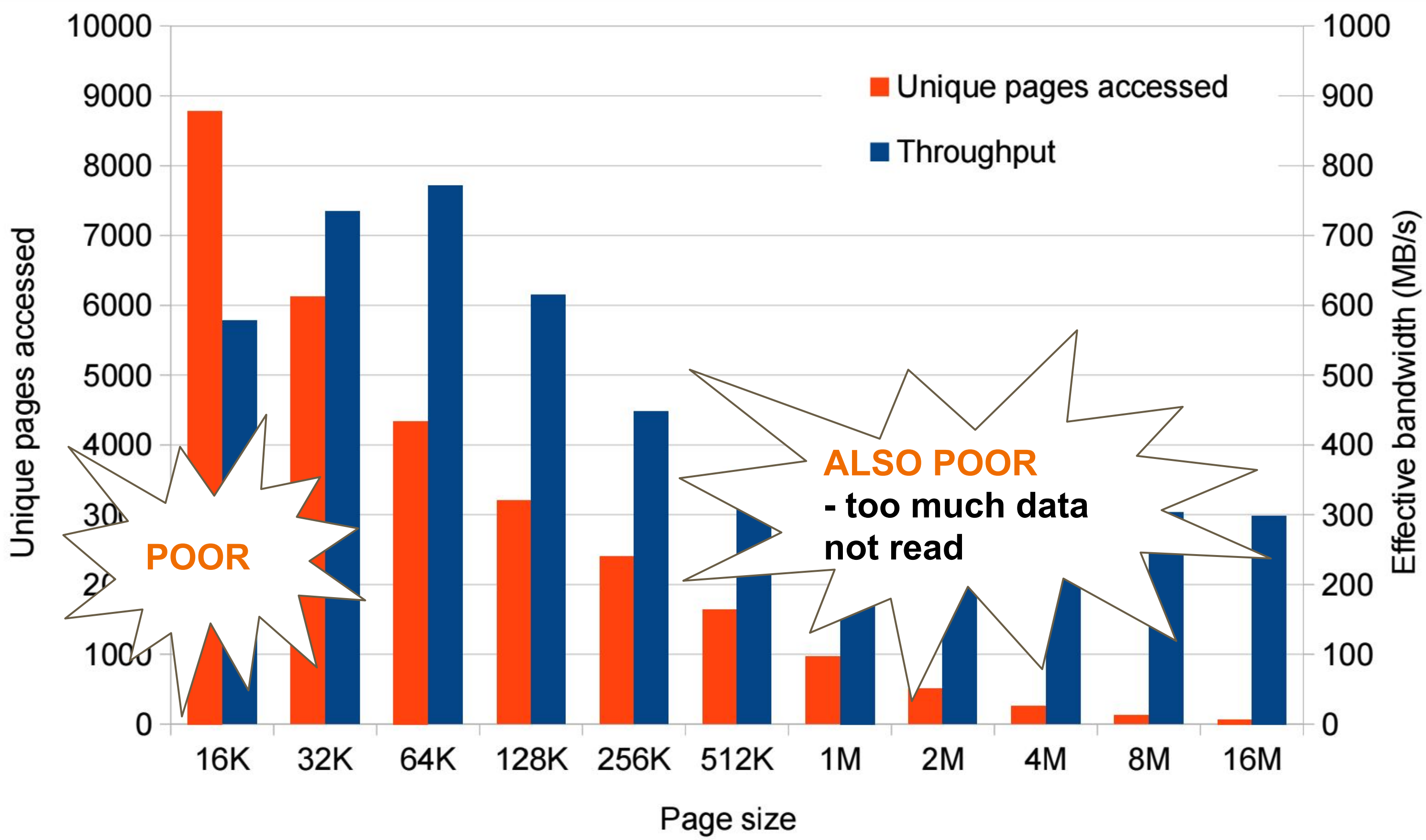


Figure 6. Random read/write performance as a function of page size. Higher is better.

5. Evaluation - Random File Read

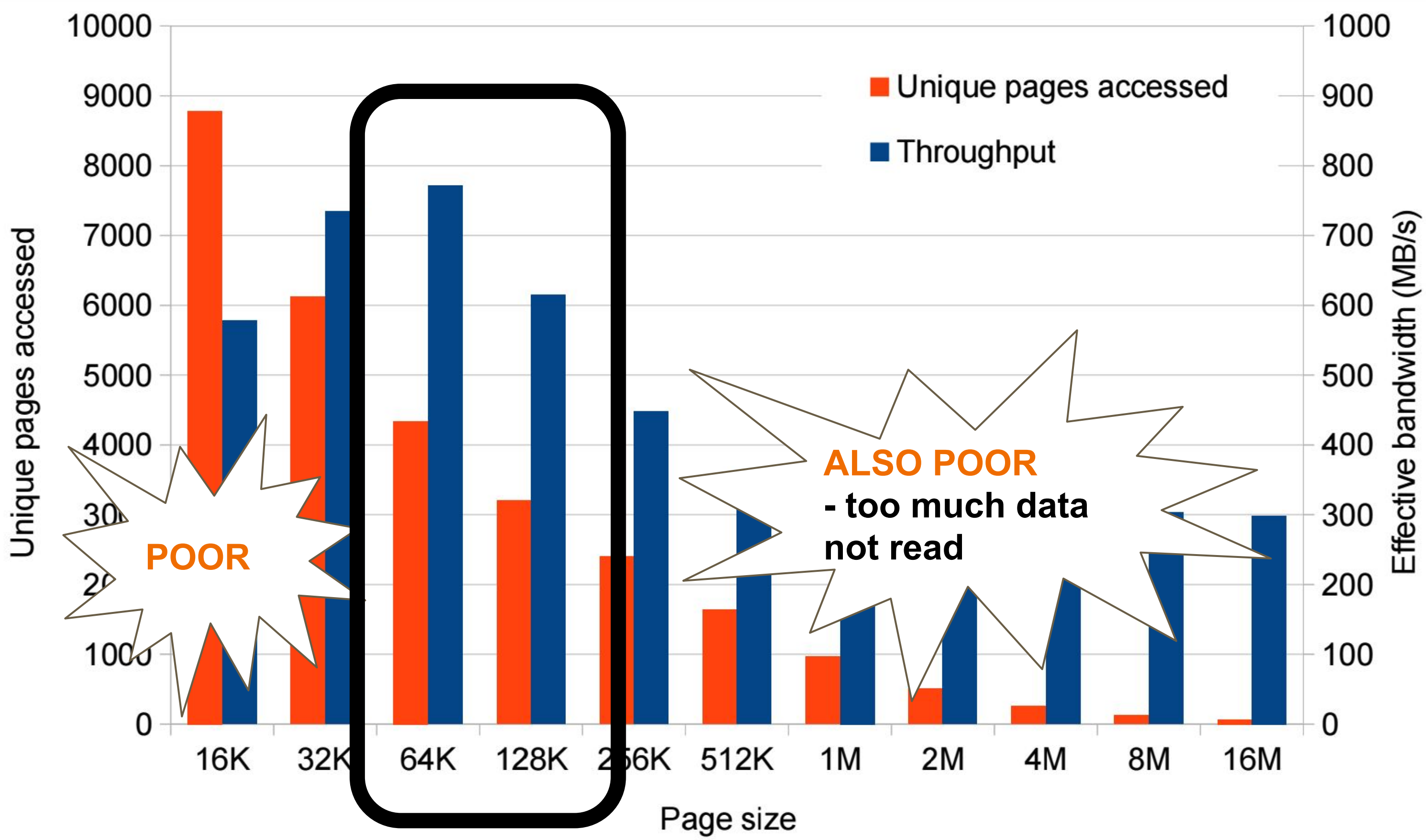
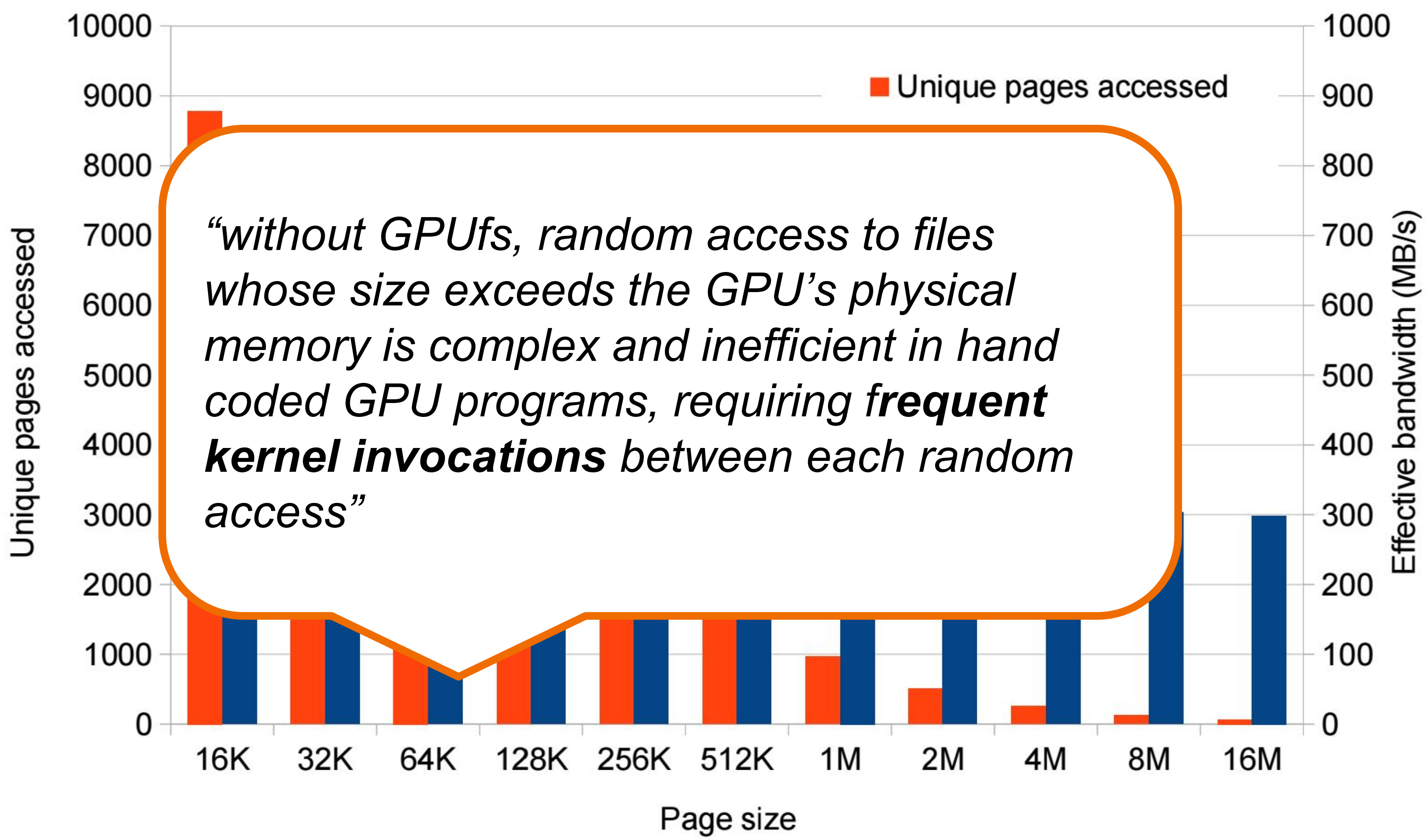


Figure 6. Random read/write performance as a function of page size. Higher is better.

5. Evaluation - Random File Read



*"without GPUfs, random access to files whose size exceeds the GPU's physical memory is complex and inefficient in hand coded GPU programs, requiring f**requent kernel invocations** between each random access"*

Figure 6. Random read/write performance as a function of page size. Higher is better.

5. Evaluation - Buffer Cache Access Performance

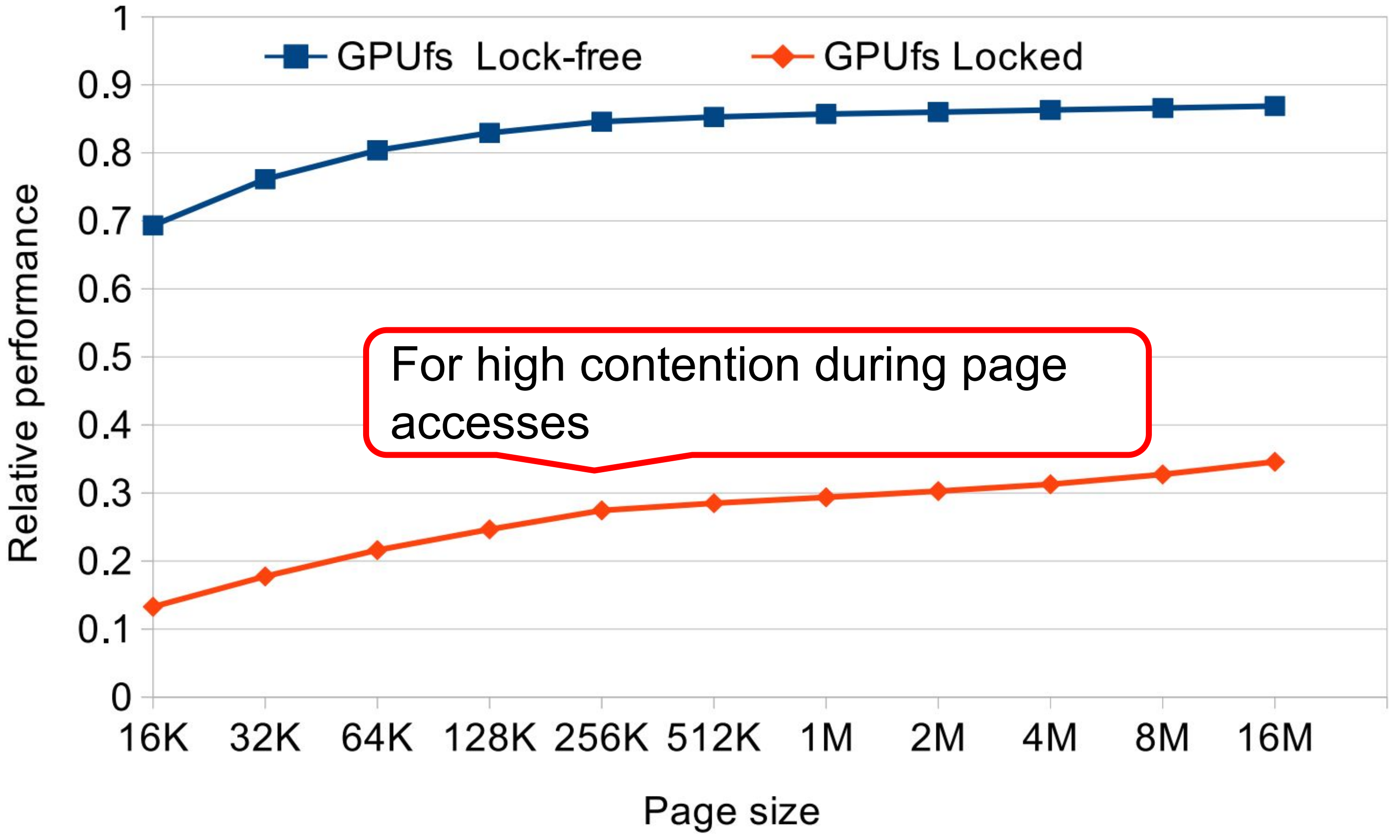


Figure 7. Buffer cache access performance with and without lock-free radix tree traversal, normalized by the raw memory access time.

5. Evaluation - Buffer Cache Access Performance

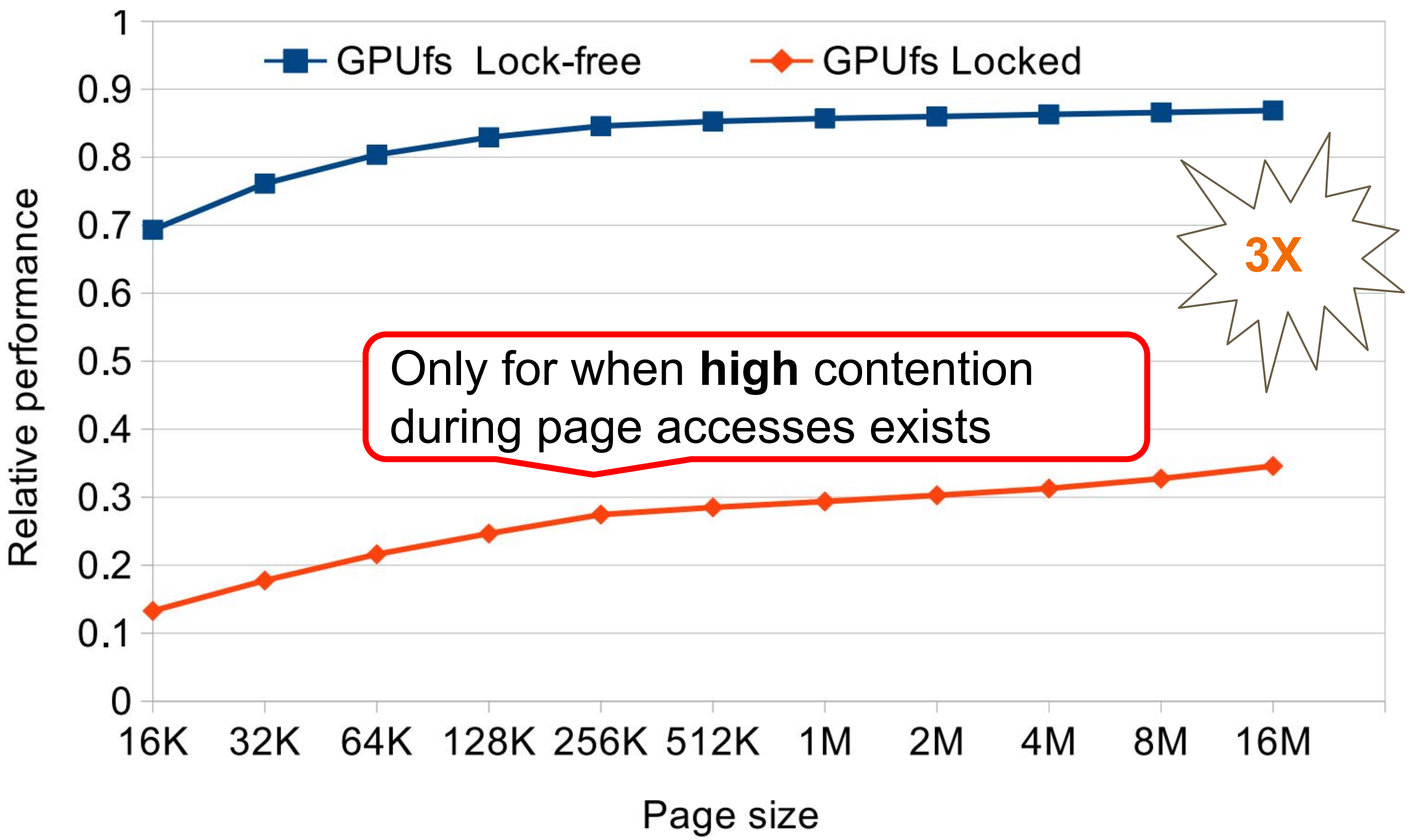


Figure 7. Buffer cache access performance with and without lock-free radix tree traversal, normalized by the raw memory access time.

5. Evaluation - Matrix-Vector Product

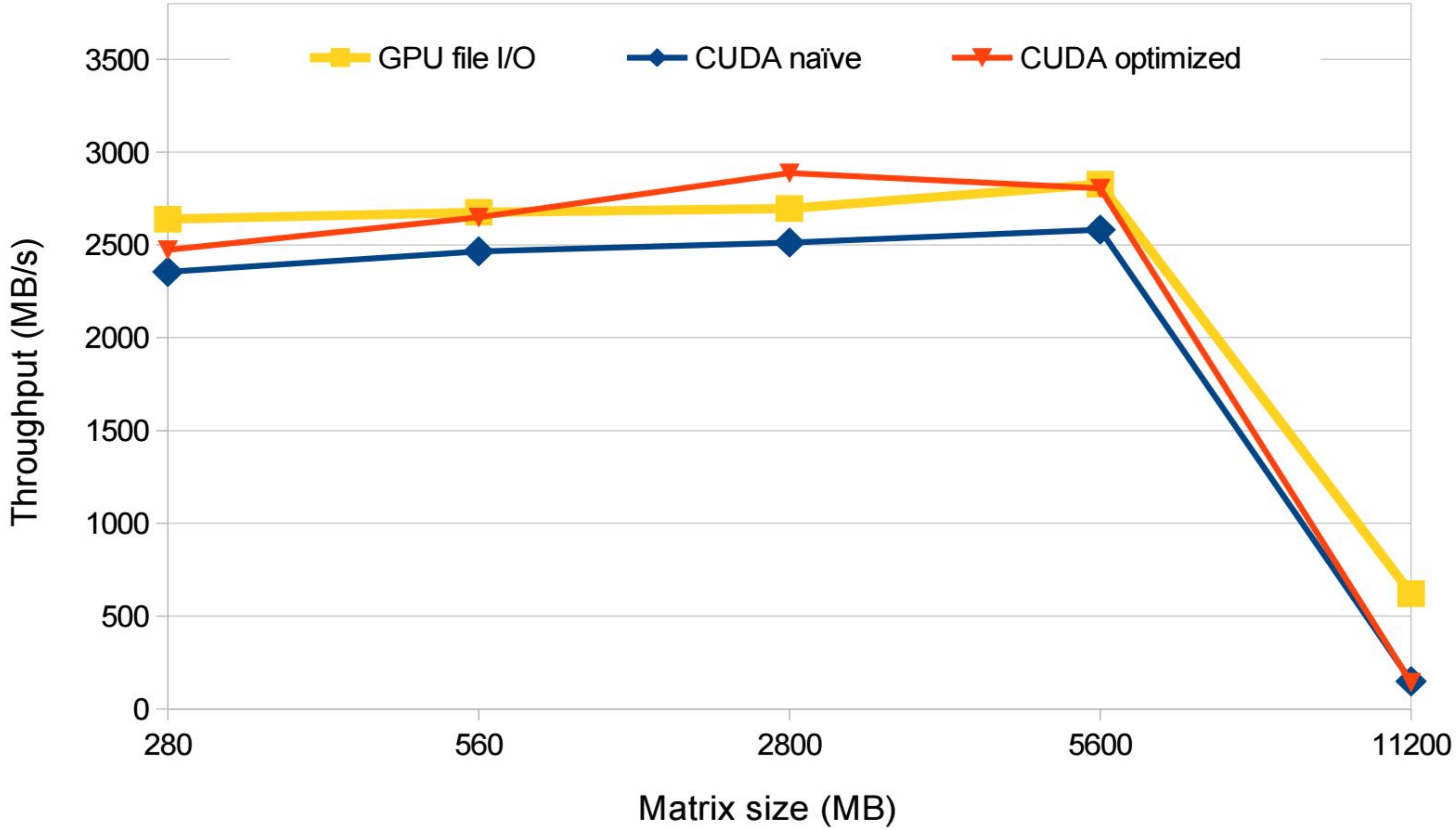


Figure 8. Matrix-vector product for large matrices

5. Evaluation - Matrix-Vector Product

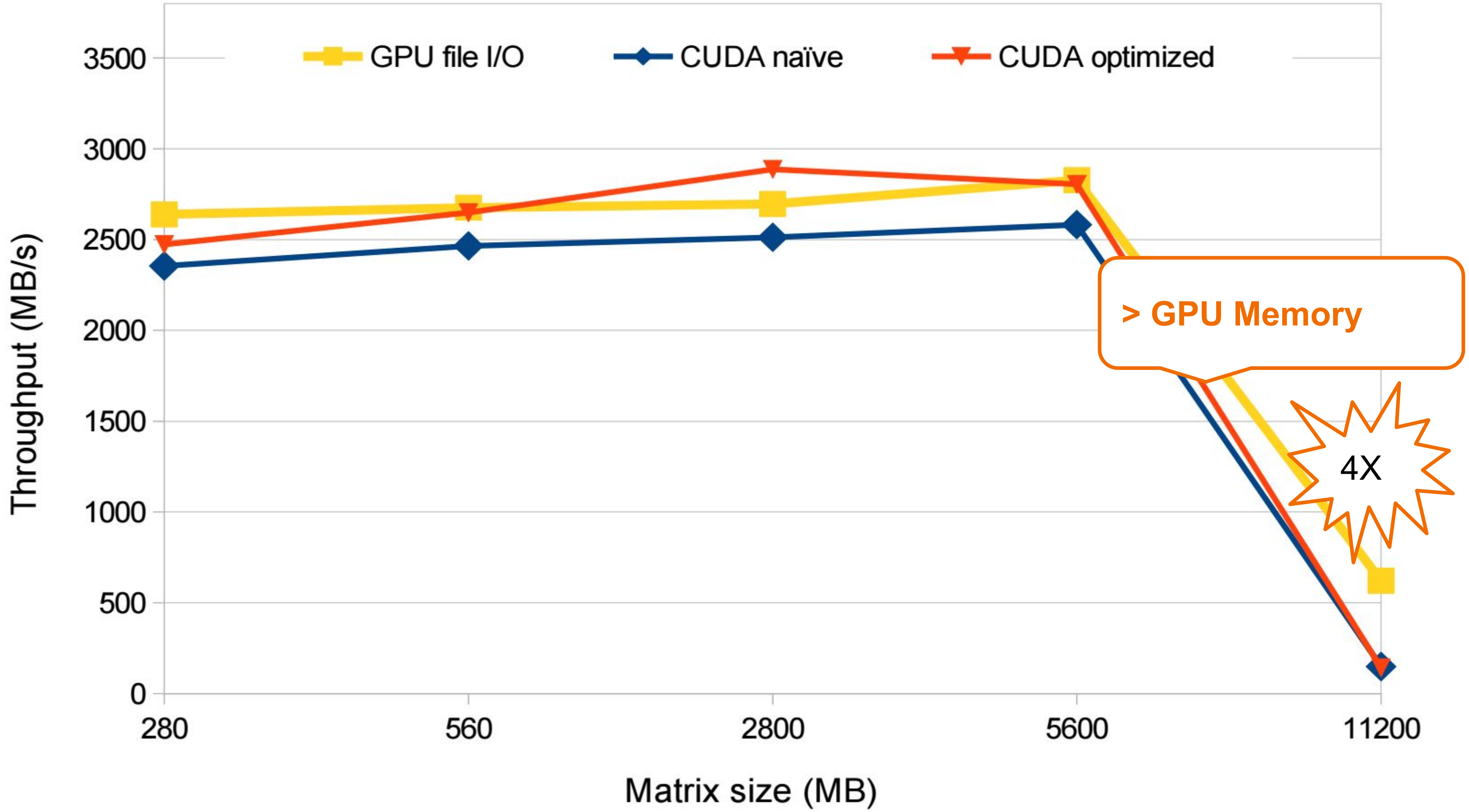


Figure 8. Matrix-vector product for large matrices

5. Evaluation - Image Search & Text Search

Buffer cache size	Time (s)	Pages reclaimed	Lock-free accesses	Locked accesses
2G	53	0	1,088,838	21,516
1G	69	11,509	547,819	574,463
0.5G	99	38,317	176,758	1,351,903

Table 2. Impact of the buffer cache size on the running time and locking behavior for the image search workload. Locked access count also includes unlocked retries.

- Find the databases that contain an image that is within a threshold of similarity w.r.t a reference image
  - **Predefined order** | Find **first match** only
  - **Random**ly generated images | Conditions with **no matches**

5. Evaluation - Ima ... xt Search

Freeing pages that were used

Buffer cache size	Time (s)	Pages reclaimed	Lock-free accesses	Locked accesses
2G	53	0	1,088,838	21,516
1G	69	11,509	547,819	574,463
0.5G	99	38,317	176,758	1,351,903

Table 2. Impact of the buffer cache size on the running time and locking behavior for the image search workload. Locked access count also includes unlocked retries.

- Find the databases that contain an image that is within a threshold of similarity w.r.t a reference image
  - **Predefined order** | Find **first match** only
  - **Random**ly generated images | Conditions with **no matches**

5. Evaluation - Image Search & Text Search

Distributed across 4 GPUs

Input	CPUx8	#GPUs			
		1	2	3	4
No match	119s	53s	27s (2.0×)	18s (2.9×)	13s (4.1×)
Exact match	100s	40s	21s (1.9×)	14s (2.9×)	11s (3.6×)

Table 3. Approximate image matching performance. Speedup for multi-GPU runs relative to a single GPU are given in parentheses.

5. Evaluation - Image Search & Text Search

Input	CPUx8	GPU-GPUfs	GPU-vanilla
Linux source	6.07h	53m (6.8×)	50m (7.2×)
Shakespeare	292s	40s (7.3×)	40s (7.3×)
LOC (semicolon)	80	140 (+52)	178

Table 4. GPU exact string match "grep -w" performance.

- Grep -w style **matching**
- Words are **short** - one word per thread
- Output buffers become **unbounded**
- Count **frequency** of word in dataset
- Frequent calls to **gopen** and **gclose**

5. Evaluation - Image Search & Text Search

OpenMP

Input	CPUx8	GPU-GPUfs	GPU-vanilla
Linux source	6.07h	53m (6.8×)	50m (7.2×)
Shakespeare	292s	40s (7.3×)	40s (7.3×)
LOC (semicolon)	80	140 (+52)	178

Table 4. GPU exact string match "grep -w" performance.

PREFETCHED

- Grep -w style **matching**
- Words are **short** - one word per thread
- Output buffers become **unbounded**
- Count **frequency** of word in dataset
- Frequent calls to **gopen** and **gclose**

5. Evaluation - Image Search & Text Search

Without GPUfs

Input	CPUx8	GPU-GPUfs	GPU-vanilla
Linux source	6.07h	53m (6.8×)	50m (7.2×)
Shakespeare	292s	40s (7.3×)	40s (7.3×)
LOC (semicolon)	80	140 (+52)	178

Table 4. GPU exact string match "grep -w" performance.

PREFETCHED

- Grep -w style **matching**
- Words are **short** - one word per thread
- Output buffers become **unbounded**
- Count **frequency** of word in dataset
- Frequent calls to **gopen** and **gclose**

5. Evaluation - Image Search & Text Search

GPUfs

Input	CPUx8	GPU-GPUfs	GPU-vanilla
Linux source	6.07h	53m (6.8×)	50m (7.2×)
Shakespeare	292s	40s (7.3×)	40s (7.3×)
LOC (semicolon)	80	140 (+52)	178

Table 4. GPU exact string match "grep -w" performance.

- Grep -w style **matching**
- Words are **short** - one word per thread
- Output buffers become **unbounded**
- Count **frequency** of word in dataset
- Frequent calls to **gopen** and **gclose**

6. Related Work

- GPUfs is the first extension of the file system abstraction to modern GPU architectures.
- However, other work exists related to technology related to individual components

Questions - I

Q. Could you talk more about the section on: Concurrent non-overlapping writes to the same file. Specifically the parts about "memory page thrashing", a single-writer MESI protocol, false sharing of buffer cache pages among different gpus, and why two copies of each cached block per GPU are needed?

Q. The underlying assumption throughout seems to be that multiple kernels access discrete parts of the same file in parallel, so they never really step on each others toes. What happens though if two separate running processes access the same location of the same file? Or say, the CPU and GPU are both accessing the same location of the same file?

Q. What is the difference between NUMA and UMA?
(see page 487)

Q. In section 3.2 under file mapping - "Improper updates to such "quasi-read-only" pages are never propagated back to the host CPU". How is this achieved if there's not a real read-only mechanism here?

A. A read-only page is never marked as dirty.

Q. In 3.4, "GPUfs is less intrusive than a complete OS because it has no active, continuously running components. " Could you please explain what are the continuously running components that are not present in GPUfs?

daemons?

Q. Section 3.4 talks about a scheduling-related weakness that makes daemon threads inefficient if running on a GPU. What exactly is the weakness? Is it the fact that it needs to be constantly running?

I think so, too. The daemon blocks could have been used for computations.

Q. Because CPUs and GPUs share the same I/O buffer in GPUfs architecture, prioritizing between different jobs and CPUs/GPUs would be crucial. How do you think if it's worth it for this tradeoff?

Q. "There is no guarantee that gmmap will map the entire file region the application requests—instead it may map only a prefix of the requested region, and return the size of the successfully mapped prefix." How is this a more efficient implementation than mmap in terms of access time?

Questions - II

Q. File operations are done at warp, rather than thread granularity, is this done so as to avoid thread divergence?

Q. File operations are done at warp, rather than thread granularity, is this done so as to avoid thread divergence?

Parallel invocation of the GPUfs API is supported at **thread block and not warp** granularity

- Simplicity and minimal divergence

Q. Why is a radix tree structure relevant in the buffered caches? Is it not better to have a hash-table based memory structure to achieve higher storage density?

Q. Why is a radix tree structure relevant in the buffered caches? Is it not better to have a hash-table based memory structure to achieve higher storage density?

- Radix trees have excellent memory usage characteristics and optimal search time characteristics - only store edges related to bit differences
- Hash tables require more memory for table loads. Collisions require handling too.

Q. In 5.1.2, it seems to say the throughput of 310MB/s isn't bad. But I am not convinced why the number is promising because it is not compared to anything.

Q. In 5.1.2, it seems to say the throughput of 310MB/s isn't bad. But I am not convinced why the number is promising because it is not compared to anything.

I think the comparison trying to be made here is that GPU code without GPUfs would typically have a throughput of 310MB/s (1/10th of 3100MB/s) , roughly equal to the worst performance seen by GPUfs.

Q. In 5.1.4, it says "the GPUfs buffer cache is sized to 2GB, with 2MB pages." With a different page size chosen, will the throughput improve for some matrix size?

Q. In 5.1.4, it says "the GPUfs buffer cache is sized to 2GB, with 2MB pages." With a different page size chosen, will the throughput improve for some matrix size?

Could possibly show marginal improvement for small increase in page sizes, however greater accuracy in this example is achieved because of multiple 2MB page reads rather than larger chunks being read (causing spurious paging of the CPU buffer, stalling CPU-GPU comms).

Thank You