GPU Scheduling

Accelerated Systems 046278 - EE Technion

Amir Watad
Streams

• We know how to run multiple kernels in the GPU.

• Streams have shortcomings though:
  
  • We have no control over scheduling algorithm.

  • No preemption
    
    • Preemption actually supported, but for graphics only, and not exposed to programmer.

  • Not a big deal. Programmer can right code carefully. (?)

• What if we want to shared the GPU between multiple processes?
PTask (SOSP ’11)

PTask: Operating System Abstractions To Manage GPUs as Compute Devices

Christopher J. Rossbach
Microsoft Research
crossbac@microsoft.com

Jon Currey
Microsoft Research
jcurrey@microsoft.com

Mark Silberstein
Technion
marks@cs.technion.ac.il

Baishakhi Ray
University of Texas at Austin
bray@cs.utexas.edu

Emmett Witchel
University of Texas at Austin
witchel@cs.utexas.edu
GPU as a compute resource

• OS can efficiently manage multiple processes on the CPU.

• GPU is just another kind of compute resource.

• Well… OS doesn’t see it so. For OS it is an I/O device.

• PTask: Let’s fix that.
The sad state of GPU integration in OS
Why OS can help?

- OS is aware of the whole system.
  - Fairness
  - Performance Isolation
  - Inter-process communication
Example Application
Example Application

- Serves multiple applications
- High frequency
- Latency is critical

==> GPU is a must
The Linux Way

cat /dev/camera | transform | filter | makeinput
Problem 1: Data Movements

1. copy camera output to user space.
2. perform transform on GPU
3. copy result back to OS.
4. copy again to user space
5. perform filter on GPU
6. copy back to OS
7. copy again to user space.
   etc..

A lot of data movements.
Problem 2: Scheduling

- GPU performing some compute
- At the same time, we want to capture mouse movements
GPU work -> System pause

Figure 4: The effect of GPU-bound work on CPU-bound tasks. The graph shows the frequency (in Hz) with which the OS is able to deliver mouse movement events over a period of 60 seconds during which a program makes heavy use of the GPU. Average CPU utilization over the period is under 25%.
Although we don’t need a lot of CPU. (Only for kernel launch)

Figure 5: The effect of CPU-bound work on GPU-bound tasks. H→D is a CUDA workload that has communication from the host to the GPU device, while H←D has communication from the GPU to the host, and H↔D has bidirectional communication.
The PTask Dataflow

• PTask handles a task dataflow

• An acyclic graph of ptasks.

• ptask = GPU program, CPU program, …

• edges = data flow (connect output of a ptask to input of another)
Design Goals

• Single resource manager => fairness and performance isolation.

• Simple programming model (hide complexities of device management, memory movements)

• Modular AND fast programs.
  • small programs which communicate efficiently.
Efficiency vs. Modularity

```java
matrix gemm(A, B) {
    matrix res = new matrix();
    copyToDevice(A);
    copyToDevice(B);
    invokeGPU(gemm_kernel, A, B, res);
    copyFromDevice(res);
    return res;
}
matrix modularSlowAxBxC(A, B, C) {
    matrix AxB = gemm(A, B);
    matrix AxBxC = gemm(AxB, C);
    return AxBxC;
}
matrix nonmodularFastAxBxC(A, B, C) {
    matrix intermed = new matrix();
    matrix res = new matrix();
    copyToDevice(A);
    copyToDevice(B);
    copyToDevice(C);
    invokeGPU(gemm_kernel, A, B, intermed);
    invokeGPU(gemm_kernel, intermed, C, res);
    copyFromDevice(res);
    return res;
}
```
Decoupling data flow from algorithm

PTask runtime sees that input and output are on same GPU no need to move data
Limitations of PTask’s graph
Limitations of PTa$k’s graph

• PTa$k assumes graph is:
  • Static (known at launch time)
  • DAG (no loops in graph)
Limitations of PTask’s graph

- PTask assumes graph is:
  - Static (known at launch time)
  - DAG (no loops in graph)
- Cannot do recursion
- Or loops (except if unrolled)
- Or graphs that are dynamic
API: Terminology

• ptask: vertex in the graph. Analogous to .... in CPU

• port: an object that can be bound to input/output of a ptask. Analogous to ?

• channel: a connection between two ports. Analogous to ?

• graph: collection of ptasks with their ports which are connected via channels. Multiple graphs can exist in parallel.

• datablocks: data units flowing in the graph

• template: describes the structure of the data
API: Terminology
Datablocks

- An abstraction of “data buffer”

- Hides location of data (CPU memory / GPU memory). Moves data between devices as needed.
Nice thing about data flow:
Concurrency is implicit
Scheduling - Problems

- Given a task graph, which tasks to execute first?
- Executing the wrong tasks might delay the whole graph.
Scheduling - Problems

- Multiple processes competing for the GPU.
- OS has no idea what a GPU is. Scheduling on the GPU is managed by the GPU driver. No body to take care of fairness.
Problem: Preemption

- OS Scheduler uses time-slice based approach
  - Preempts a task after a while to schedule another task
- GPUs do not support preemption (except for graphics)
Scheduling Policy

• First available: All ready tasks compete on the GPU
• FIFO: All ready tasks are queued
• Priority: Tasks are assigned priorities
• Data-aware: …
Priorities

- static priority for each task
- proxy priority: priority of the CPU thread managing the ptask.
void schedule() {
    update_eff_prio(ptasks);
    sort(ptasks); // by descending eff prio
    while(gpus_available() && size(ptasks)>0) {
        foreach(p in ptasks) {
            best_gpu = match_gpu(p);
            if(best_gpu != null) {
                remove(ptasks, p);
                p->dispatch_gpu = best_gpu;
                signal_dispatch(p);
                return;
            }
        }
    }
}
Priority Scheduling

```c
void update_eff_prio(ptasks) {
    avg_gpu = avg_gpu_time(ptasks);
    avg_cwait = avg_current_wait(ptasks);
    avg_dwait = avg_decayed_wait(ptasks);
    avg_pprio = avg_proxy_prio(ptasks);
    foreach(p in ptasks) {
        boost = W_0*(p->last_cwait - avg_cwait);
        boost += W_1*(p->avg_wait - avg_dwait);
        boost += W_2*(p->avg_gpu - avg_gpu);
        boost += W_3*(p->proxy_prio - avg_pprio);
        p->eff_prio = p->prio + boost;
    }
}
```
Priority Scheduling

gpu match_gpu(ptask) {
    gpu_list = available_gpus();
    remove_unfit(gpu_list, ptask);
    sort(gpu_list); // by descending strength
    return remove_head(gpu_list);
}
PTask Scheduler: Work conserving?

- Work conserving: Tries to make resources busy as much as possible.
Data-Aware Scheduling

- Same as priority scheduling
- But selects GPU based on location of majority of inputs (rather than strength)
  - Reduces memory movements
- If priority > threshold
  - Select strongest GPU if preferred GPU not available
  - Else: wait until preferred GPU is available
PTask in Linux

- Full PTask implementation was for windows.

- In linux: make GPU a real citizen of the OS (managed by the OS itself rather than PTask subsystem)

- GPU accounting is task_struct

- One system call: tell OS when a GPU is being used. (kernel start and kernel end)
  
  - Ask the user for help, because driver is closed source.
  
  - Could be done right. Later work solved this.
Enforce fairness by punishing hogs

• Once a process is handed the GPU, we cannot stop it from using it.

• Instead, we punish it next time.

  • Leaky bucket algorithm.

  • Assumptions / Limitations?
Leaky Bucket

- Each process starts with a budget of $B$ tokens.
- “Pays” when uses the resource.
- Gets a few tokens periodically.
- You can pay more than you have, but the bank will call.
Leaky Bucket

- Process \( p \) has GPU budget \( B(p) \)
- When it runs for \( t(p) \) time, budget is reduced by \( t(p) \)
- Budget incremented by \( q \) every \( T \) time.
- Negative budget: cannot use GPU. wait..
• Maximum Budget: (takes into account Linux priority \( n(p) \))

\[
B_{max_p} = \frac{n_p}{\sum_{i \in P} n_i} \sum_{i \in P} t_i
\]
Evaluation:

Figure 13: Number of ptask invocations per second on 1 GTX580 GPU for four competing PTask graphs with different ptask priorities (2, 4, 6, 8) using 6x6 rectangular graphs of matrix multiplication ptasks. The trends are the same independent of graph size, shape, or ptask composition. Data for the data-aware policy is not shown because the the priority and data-aware policies are identical when only 1 GPU is present.
Evaluation: EncFS

• Encryption file system
• Encrypts on write, decrypts on read
• implemented using FUSE
• Heavy GPU tasks running in the background.
  • Kernels are long. GPU is monopolized. No fairness. Encryption is slow.
  • But PTask solves this :)}
Evaluation: EncFS

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>1 GPU task</th>
<th>2 GPU tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Linux</td>
<td>PTSched</td>
</tr>
<tr>
<td>read</td>
<td>247 MB/s</td>
<td>-10.0×</td>
<td>1.16×</td>
</tr>
<tr>
<td>write</td>
<td>82 MB/s</td>
<td>-8.2×</td>
<td>1.21×</td>
</tr>
</tbody>
</table>

Table 3: Bandwidth measurements for sequential read or write of a 200MB file on an encrypted file system, relative to the CPU performing encryption (CPU column). Negative numbers indicate reduced bandwidth. There are one or two concurrently executing background GPU-heavy tasks, running with the default Linux scheduler (Linux) or our scheduler (PTSched).