Batch-Aware Unified Memory Management in GPUs for Irregular Workloads

Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim

GPU, Unified memory management, Memory oversubscription, Virtual memory
https://doi.org/10.1145/3373376.3378529

Abstract

While unified virtual memory and demand paging in modern GPUs provide convenient abstractions to programmers for working with large-scale applications, they come at a significant performance cost. We provide the first comprehensive analysis of major inefficiencies that arise in page fault handling mechanisms employed in modern GPUs. To amortize the high costs in fault handling, the GPU runtime processes a large number of GPU page faults together. We observe that this batched processing of page faults introduces large-scale serialization that greatly hurts the GPU's execution throughput. We show real machine measurements that corroborate our findings.
Our goal is to mitigate these inefficiencies and enable efficient demand paging for GPUs. To this end, we propose a GPU runtime software and hardware solution that (1) increases the batch size (i.e., the number of page faults handled together), thereby amortizing the GPU runtime fault handling time, and reduces the number of batches by supporting CPU-like thread block context switching, and (2) takes page eviction off the critical path with no hardware changes by overlapping evictions with CPU-to-GPU page migrations. Our evaluation demonstrates that the proposed solution provides an average speedup of 2x over the state-of-the-art page prefetching. We show that our solution increases the batch size by 2.27x and reduces the total number of batches by 51% on average. We also show that the average batch processing time is reduced by 27%.

Problem Statement and Research Objectives

- UVM provides a coherent view of a single virtual address space between CPUs and GPUs with automatic data migration via demand paging.
  - While the feature sounds promising, in reality, the benefit comes with a non-negligible performance cost.
- The goal of this work is to support the efficient execution of large-scale irregular applications, such as graph computing workloads, in the UVM model.

Previous Study

Recently, Li et al.[1] proposed a memory management framework, called **eviction-throttling-compression (ETC)**, to improve GPU performance under memory oversubscription.
→ However, for many large-scale, irregular applications, we found that the ETC framework is ineffective.

- Since irregular applications **access a large number of pages within a short period of time**, predicting correct timing is not trivial.
- For this to be effective, the working set size has to be reduced when GPU cores are **throttled**[1].
  - This is the case for most regular workloads.
  - However, this is **not the case for many large-scale**, irregular applications because **most of the memory pages are shared across GPU cores**, and thus, memory-aware throttling is not effective in reducing the working set size.

Background

Thread Concurrency in GPUs

GPU shader cores, such as NVIDIA Streaming Multiprocessor (SM), AMD Compute Unit (CU), or Intel Execution Unit (EU), provides hardware resources that are required to keep the contexts of multiple threads without doing conventional context switching.

- For example in NVIDIA GPUs, the maximum concurrency is capped by the maximum number of threads and thread blocks (e.g., 2048 and 32, respectively), the register file size (e.g., 64k 32-bit registers), and the maximum number of registers per thread (e.g., 255), among others.

Unified Virtual Memory in GPUs

- Virtual Memory
  - To translate a virtual address into a physical address, the GPU performs a **page table walk**.
    1. To accelerate this, translation lookaside buffers (TLBs) are adopted from CPUs and optimized for GPUs.
    2. Requiring a commensurate number of translations → **highly threaded page table walker**
    3. A multilevel page table requires many memory accesses to translate a single address → **page walk cache**
- Demand Paging

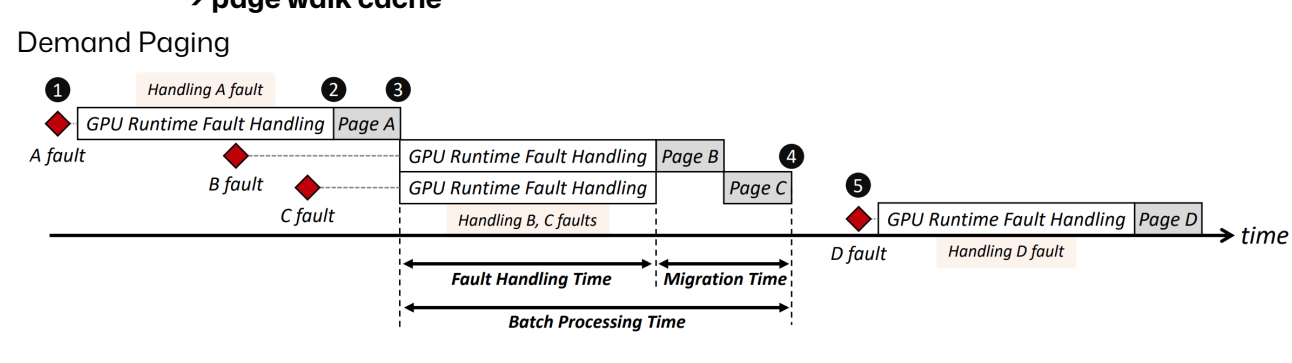

Figure 2. Overview of how GPU page faults are handled by the GPU runtime.

- GPU runtime fault handling time: To handle a multitude of page faults efficiently, the GPU runtime preprocesses the page faults before performing page table walks.
  - This preprocessing includes sorting the page faults in ascending order of page addresses (to accelerate the page table walks) and the analysis of page addresses to insert page prefetching requests.
- Batch processing time: The time between the beginning of a batch's processing and the migration of the last page.
  - When batch processing ends, the GPU runtime checks whether there are waiting page faults (pages B and C in the figure). Then, the GPU runtime begins to process them immediately.

1. Batch processing time

The batch processing time is measured to be in the range of 223μs to 553μs with a median of 313μs, of which GPU runtime fault handling accounts for an average of 46.69% of the time (measured to be in the range of 50μs to 430μs with a median of 140μs).

- To amortize the GPU runtime fault handling time, it can be obtained by increasing the batch size (i.e., the number of page faults handled together in a batch)

> **Note**
> **Prior work**[2] reports that page fault handling latency ranges from 20μs to 50μs, but these numbers are conservative and can be worse depending on the applications and systems.

2. Unnecessary serialization

Page evictions introduce unnecessary serialization in page migrations.

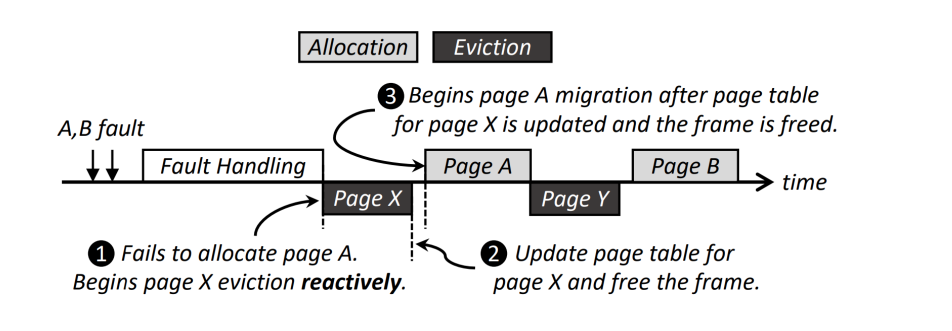

Figure 6. Overview of how when GPU runtime evicts a page from GPU memory, and why it is on the critical path.

- page evictions and new page allocations are serialized in modern GPUs to prevent the new pages from overwriting the evicted pages.
  - alloc_root_chunk(): The physical memory allocator in the GPU runtime tries to allocate a new page
  - pick_and_evict_root_chunk(): If it failed, page eviction is requested
  - chunk_start_eviction(): Once victim is selected, its eviction flag is set.
  - evict_root_chunk(): The eviction begins

Proposed Method

1. Thread Oversubscription (TO)

- a **CPU-like thread block context switching** technique, to effectively amortize the GPU runtime fault handling time by **increasing the batch size** (i.e., the number of page faults handled together)
  - [Only up to 64 **warps**[3] (or 2048 threads) can concurrently run in SM] + [A warp is stalled once it generates a page fault] → [it does not take much time before the GPU becomes crippled due to lack of runnable warps]
  - In the presence of page migration between CPU and GPU memory, increasing thread concurrency is beneficial despite the expensive context switching overhead.

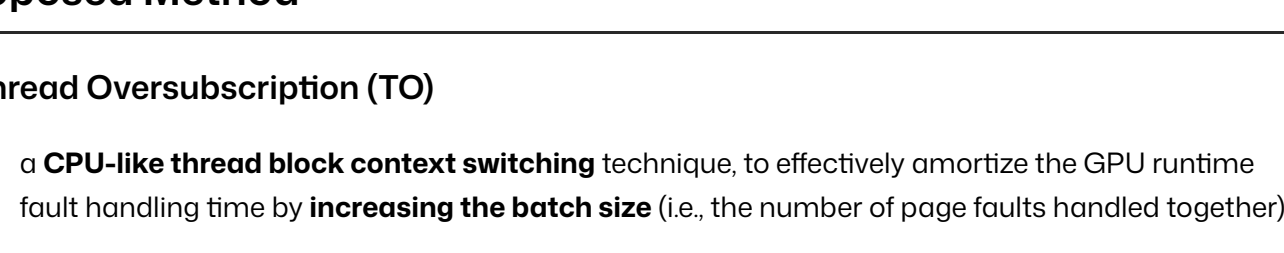

Figure 7. Overview of how thread oversubscription can increase the batch size. TB is short for thread block.

> The GPU runtime fault handling time for the second batch is slightly increased to handle an additional page fault for page D.

- We develop thread oversubscription, a GPU virtualization technique. We utilize the **Virtual Thread (VT)**[4] as our baseline architecture for GPU virtualization, → **an extent of VTC**
  1. We employ an **additional mapping table** so that different **Virtual Warp IDs (VWIs)**[5] can access the same set of register files when they are context switched.
  2. We extend the operation performed by the **Virtual Thread Controller (VTC)**[6]
     - Baseline VT only stores the per-thread block state information in the shared memory through the context handler
     - We extend this operation to **store register files as well**
  3. We **dynamically control the degree of thread oversubscription** based on the rate at which **premature eviction**[7] occurs.

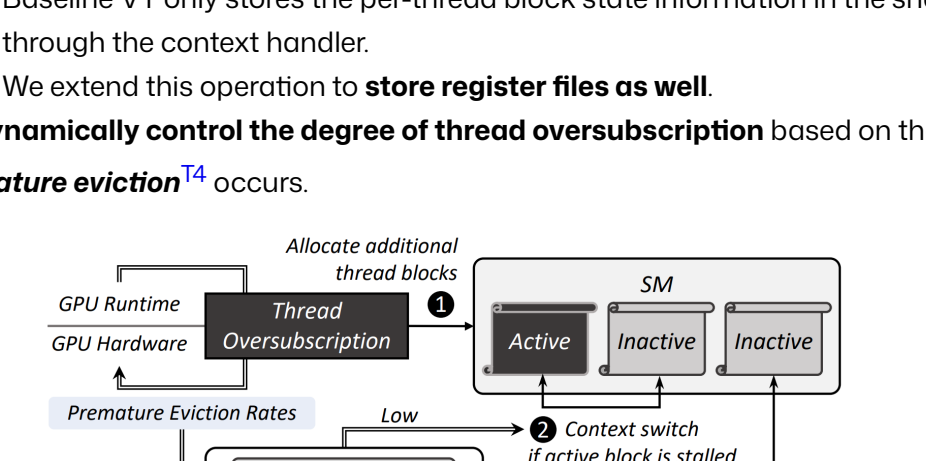

Figure 8. Thread oversubscription scheme.

2. Unobtrusive Eviction (UE)

- to take GPU page evictions off the critical path with no hardware changes based on the idea of overlapping page evictions with CPU-to-GPU page migrations.

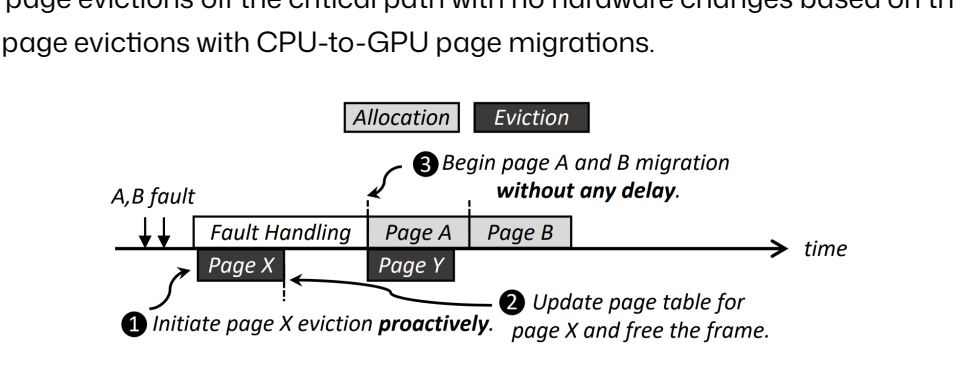

Figure 10. Overview of how unobtrusive eviction works.

> At the same time, page Y can be evicted using bidirectional transfers.

- There is an opportunity to reduce the page migration time itself
  1. baseline: the performance of a GPU with 50% memory oversubscription
  2. unlimited: the performance of a GPU with unlimited memory where no page evictions occur
  3. ideal eviction: the performance of a GPU with an instant page eviction capability (= overhead of eviction is 0)

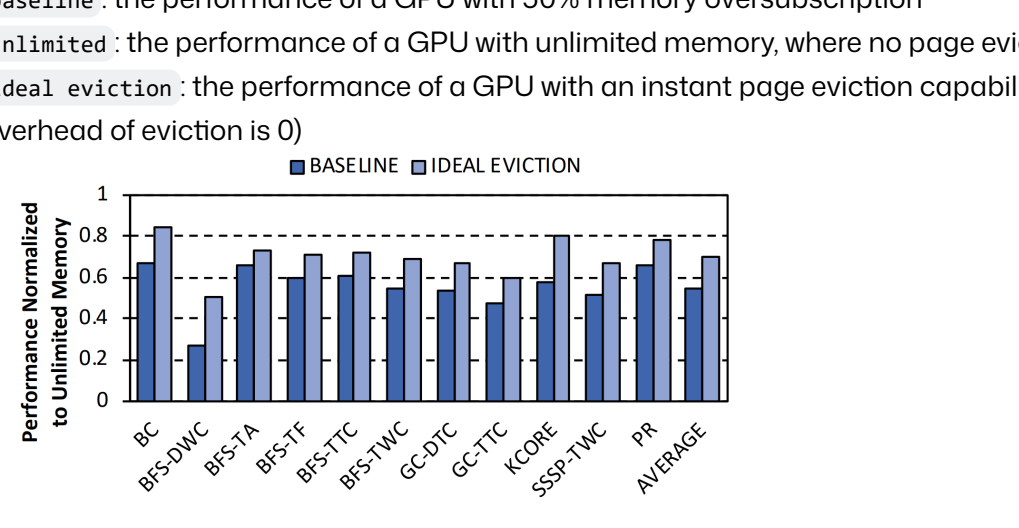

Figure 8. Performance of a GPU with 50% memory oversubscription compared to a GPU with unlimited memory, and how the performance changes with ideal eviction.

> baseline vs. unlimited: average performance loss of 46% in baseline
> baseline vs. ideal eviction: average performance improvement of 16% in ideal eviction.

- Our goal is to devise a mechanism that **exploits bidirectional transfers** without violating the serialization requirement.
  - The key idea is to **preemptively initiate a single page-➊ eviction** and **enable pipelined bidirectional transfers afterwards**.
  - To perform this preemptive eviction promptly at the beginning of batch processing, we modify the GPU runtime and **add a new GPU memory status tracker**.

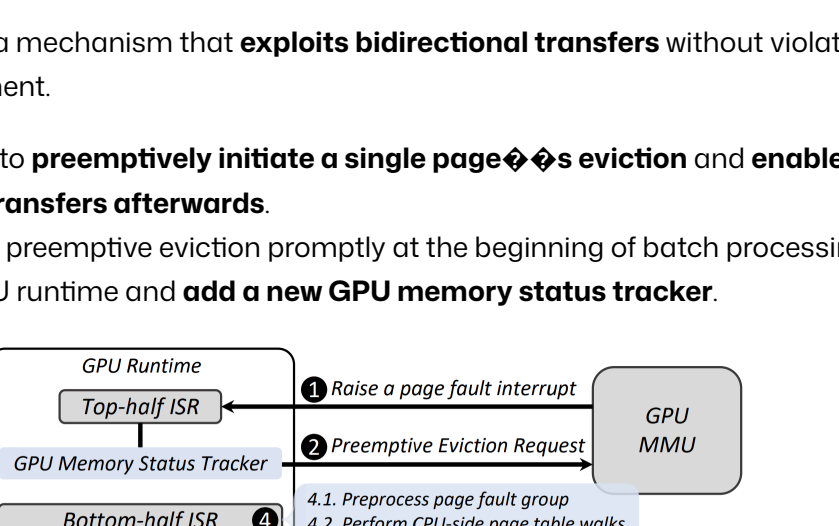

Figure 9. Unobtrusive eviction scheme.

Evaluation and Results

1. Simulator

We use **MacSim**[a], a cycle-level microarchitecture simulator. We modify the simulator to support virtual memory, demand paging, and the **Virtual Thread (VT)**[4]

2. Workloads

We select 11 workloads from the **GraphBIG**[b] benchmark suite.

- BC(Betweenness Centrality) is an algorithm that detects the amount of influence a node has over the flow of information in a graph.
- Graph traversal(BFS(Breadth First Search)) is the most fundamental operation of graph computing, for which we include five different implementations
  - data-warp-centric (DWC)
  - topological-atomic (TA)
  - topological-frontier (TF)
  - topological-thread-centric (TTC)
  - topological-warp-centric (TWC)
- GC(Graph Coloring) performs the assignment of labels or colors to the graph elements (i.e., vertices or edges) subject to certain constraints, for which we include two different implementations
  - data-thread-centric (DTC)
  - topological-thread-centric (TTC)
- KCORE(K-core Decomposition) partitions a graph into layers from external to more central vertices.
- SSSP(Single-Source Shortest Path) finds the shortest path from the given source to each vertex, for which we include a topological-warp-centric (TWC) implementation.
- PR(Page Rank) is an algorithm that evaluates the importance of web pages.

3. Performance Comparison

- **BASELINE**[7]
- **ETC**[1]

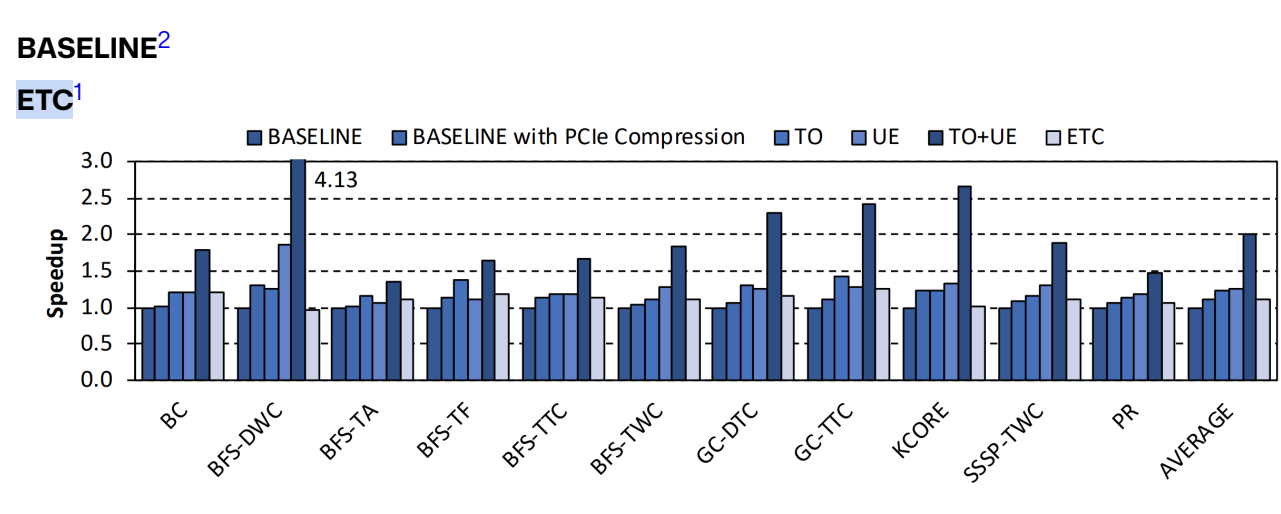

Figure 11. Performance comparison among baseline with the state-of-the-art page prefetching [10] (with and without PCIe compression), eviction-throttling-compression (ETC) [39], and our proposed mechanisms (thread oversubscription is denoted as TO and unobtrusive eviction is denoted as UE), all normalized to baseline.

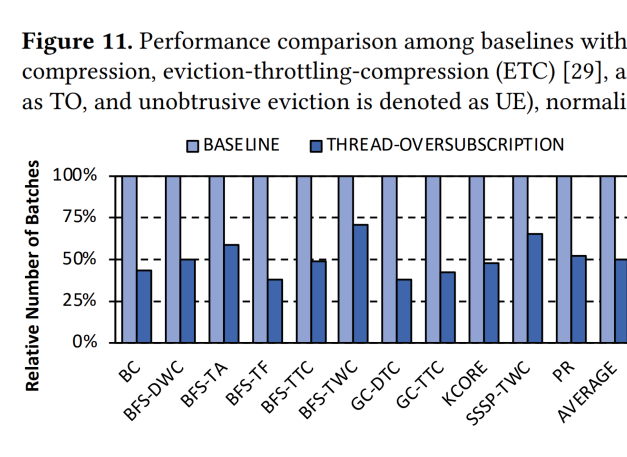

Figure 12. Total number of batches.

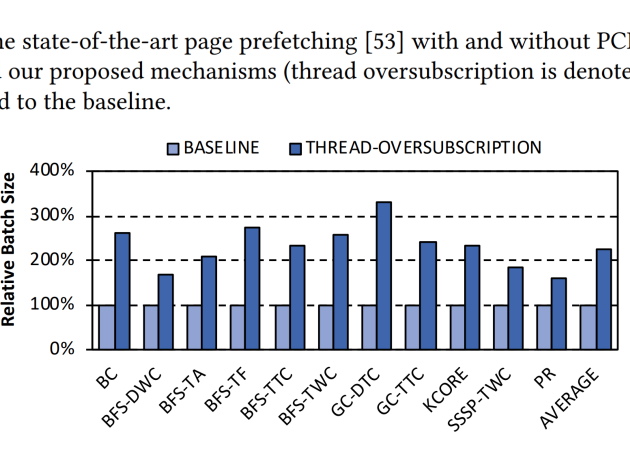

Figure 13. Average batch sizes.

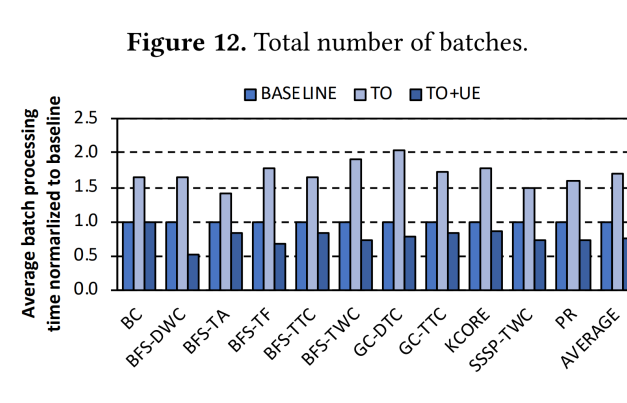

Figure 14. Average batch processing time.

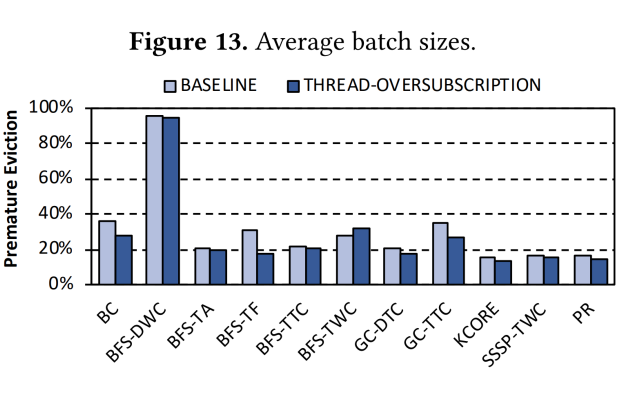

Figure 15. Premature eviction comparison.

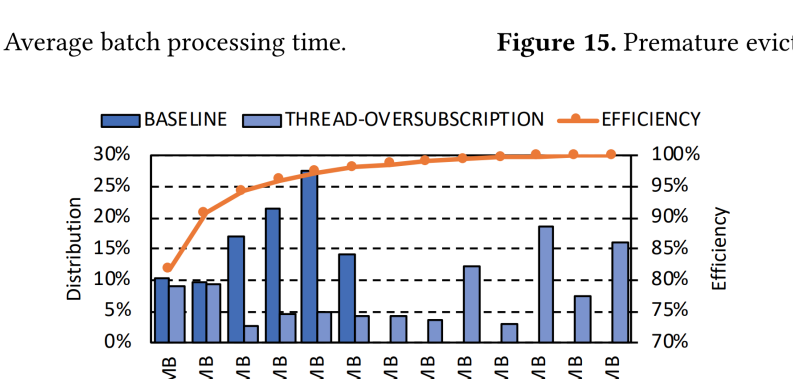

Figure 16. Batch size comparison.

Notes

Terminology

1. **Dynamic frequency scaling (also known as CPU throttling)** : a power management technique in computer architecture whereby the frequency of a microprocessor can be automatically adjusted "on the fly" depending on the actual needs, to conserve power and reduce the amount of heat generated by the chip. ↩
   - https://en.wikipedia.org/wiki/Dynamic_frequency_scaling
2. **VTC(Virtual Thread Controller)** : It keeps track of the state of all thread blocks in order to determine which inactive thread blocks can be brought back from the inactive to an active state, or vice versa, when a thread block is swapped out. ↩
3. **VWI(Virtual Warp ID)** : It is a unique warp identifier across all the assigned warps to an SM, including both active and inactive thread blocks. Only when a thread block finishes execution are its VWIs released and reused for another thread block. ↩
4. **Premature eviction** : It occurs when a page is evicted earlier than it should be, and a page fault is generated for the page again by the GPU. ↩
5. **Warp** : the primary execution unit in GPUs. It is a collection of scalar threads (e.g., 32 in NVIDIA GPUs) that run in a single-instruction multiple-thread (SIMT) fashion. ↩
6. **TLP(Thread Level Parallelism)**
7. **ISR(Interrupt Service Routine)**

1. [29] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. "A Framework for Memory Oversubscription Management in Graphics Processing Units." In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ↩
   - https://doi.org/10.1145/3297858.3304044
2. [53] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. 2016. "Towards High Performance Paged Memory for GPUs." In Proceedings of the International Symposium on High Performance Computer Architecture (HPCA). ↩
   - https://doi.org/10.1109/HPCA.2016.7446077
3. [52] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram. 2016. "Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit." In Proceedings of the International Symposium on Computer Architecture (ISCA). ↩
   - https://doi.org/10.1109/ISCA.2016.59
4. [27] Hyesoon Kim, Jaekyu Lee, Nagesh B. Lakshminarayana, Jaewoong Sim, Jieun Lim, Tri Pho, Hyojong Kim, and Ramyad Hadidi. 2012. "MacSim: A CPU-GPU Heterogeneous Simulation Framework User Guide." ↩
   - https://scholar.googleusercontent.com/scholar?q=cache:...
5. [37] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. "GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions." In International Conference for High Performance Computing, Networking, Storage and Analysis (SC). ↩
   - https://doi.org/10.1145/2807591.2807626