Design and Analysis of GPU Algorithms

by

Heng Zhao
B.Sc., The University of Sydney, 2020

An Industrial Project Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Design and Analysis of GPU Algorithms

by

Heng Zhao
B.Sc., The University of Sydney, 2020

Supervisory Committee

Dr. Sean Chester, Supervisor
(Department of Computer Science)

Dr. Ibrahim Numanagić, Departmental Member
(Department of Computer Science)

ABSTRACT

In this industrial project, we summarized the main characteristics of CPU-GPU heterogeneous system architectures. we then developed an analysis model focused on GPU parallel algorithms. Lastly, we developed two novel GPU algorithms for the decremental reachability problem with different approaches and analyzed them in our model. After a thorough analysis by algorithm designers, the results can be compared within systems to determine which algorithms are practically efficient.

Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **iv**

List of Tables **vi**

List of Figures **vii**

1 Introduction and Related Work **1**

1.1 Contributions 3

1.2 Outlines 3

2 Architecture of Heterogeneous System **4**

3 Model of Analysis **10**

4 Model Applicability **13**

5 Algorithm Design **15**

5.1 Problem setup: 15

5.2 Vertex-Centric Algorithm: 16

5.2.1 Analysis: 20

5.3 Edge-Centric Algorithm: 21

5.3.1 Analysis: 23

6 Conclusion and Future Work **25**

7 Glossary of Terms **26**

Bibliography **28**

List of Tables

Table 3.1 GPU Hardware 10
Table 3.2 GPU Computation 10

Table 5.1 Algorithm comparisons 24

List of Figures

Figure 2.1 an abstract overview of data transfer within a thread block cluster. 5
Figure 2.2 an abstract overview of data transfer from external memory to the GPU. 6
Figure 2.3 an abstract overview of GPU computation architecture showcases the context switching of a fixed number of warps into and out of one assigned Streaming processor. 8
Figure 2.4 an example of branch divergence. 9

Figure 3.1 overview of the model. 11
Figure 3.2 computation analysis example. 12

Figure 5.1 simplified example of both 5.2 & 5.3 algorithms running 19

Chapter 1

Introduction and Related Work

Nowadays, many structurally complex computational systems have been developed to run specialized algorithms. Some widely used are Tensor Processing Units (TPUs) [8] for multi-linear algebra calculations and AI accelerators such as Vision Processing Units (VPUs) [13] for machine learning computations. The most renowned would be Graphics Processing Units (GPUs) [12]. Due to their parallel architecture, GPUs have been discovered to be efficient not only for graphics-related problems. These days, many algorithms have been developed dedicated to each system for their specialized tasks. However, a significant concern that troubles system researchers and algorithm developers is how we theoretically study and analyze the algorithms running in such complicated computational systems.

The Random-Access Machine (RAM) model [11] is the most widely used for algorithm analysis, and everyone learns that from their first algorithm class. Today, many researchers, especially theoretical computer scientists, continue to use the RAM model for algorithm analysis. RAM is a concise yet powerful model for providing an asymptotic analysis of the number of computations that algorithms do. However, RAM was developed only for sequential algorithms.

Then, computer scientists realized that parallelism provides the uppermost speed-up of algorithms. So, the Parallel Random-Access Machine (PRAM) model [5] was developed with the same essence of RAM but also captures the effectiveness of scaling and parallelism in a parallel algorithm. PRAM is the most commonly used model when one wants to give some theoretical analysis of parallel algorithms in their sys-

tems. However, PRAM ignores other time-consuming operations, even in a typical single-core CPU system, such as data transfer time between external memory and caches. Consequently, PRAM will provide a loose analysis of memory-intensive algorithms in big-data systems where the memory operation becomes the bottleneck [4](p. 16) [7].

In 2010, Arge, et al[2] introduced the Parallel External Memory (PEM) model. This was a significant attempt to capture the characteristics of parallel algorithms for private-cache chip multiprocessors (CMPs) by assuming that data operations are the bottleneck. The only metric in this model is the I/O complexity. However, this model does not always apply well to GPU algorithms. Depending on the size of the problem, the input may initially be stored in higher hierarchical storage such as VRAM or RAM rather than external memory. This can result in computations becoming the bottleneck for GPU algorithms. Furthermore, the PEM model lacks features to accurately capture the complexity of numerous threads performing data operations simultaneously within GPU systems.

In 2014, J. A. Ang et al. [1] introduced Abstract Machine Models, which provide a completely different approach to the performance evaluation of algorithms. This abstract model does not define specific parameters for algorithm analysis; instead, it designs blueprints for a few typical high-performance computing architectures. One should derive additional hardware details from a blueprint to define the parameters in specific systems. This is an interesting approach but may lead to some problems. There is no fixed parameter for analyzing algorithms. Although this provides massive flexibility for fitting different architectures, it may sound good for developers. However, the algorithm designers would have to justify each algorithm's detailed assumptions for adequate applicability, which leads to less abstraction of the algorithms. Another problem is that one may get lost in too many details through experiments and end up with an over-complicated model, as shown in [1] pp. 18-20, which makes it impossible to provide any algorithm design guidelines and analysis.

One can already see that there are many factors in analyzing the behaviours of algorithms in modern systems. However, the time cost of almost all algorithms is dominated by two parts:

1. Data operations between where the problem is stored(often in external memory when the problem size is sizable) and where processors can directly retrieve from(often

in registers).

2. Computations performed by processors.

By this essence, let us summarize the data & computation architecture of the Heterogeneous System.

1.1 Contributions

1. This report concludes an abstract overview of the CPU-GPU heterogeneous system.
2. This report provides a novel asymptotic analysis model for algorithms in the heterogeneous system.
3. This report implements two novel parallelized dynamic graph algorithms in the heterogeneous system.

1.2 Outlines

1. In the next chapter 'Architecture of Heterogeneous System', we will describe the architecture so that one can understand how the system informs the model.
2. In the 'Model of Analysis' chapter, we will define and describe our analysis model for the heterogeneous system.
3. In the 'Model Applicability' chapter, we will explain design thinking and trade-offs in this model so algorithm designers can determine if the model applies to their algorithm analysis.
4. In the 'Algorithm Design' chapter, we will showcase our two novel algorithms for solving a decremental reachability problem in the heterogeneous system. We then thoroughly analyze them and give examples of choosing algorithms based on our model metrics and specific systems.
5. Lastly, in the 'Conclusion and Future Work' chapter, we summarize the report and suggest further research opportunities following this work.

Chapter 2

Architecture of Heterogeneous System

The most challenging part of constructing models of computation is maintaining their conciseness, which makes them applicable while still capturing the crucial features of the system. In this chapter, we aim to summarize important facts about CPU-GPU heterogeneous system architecture.

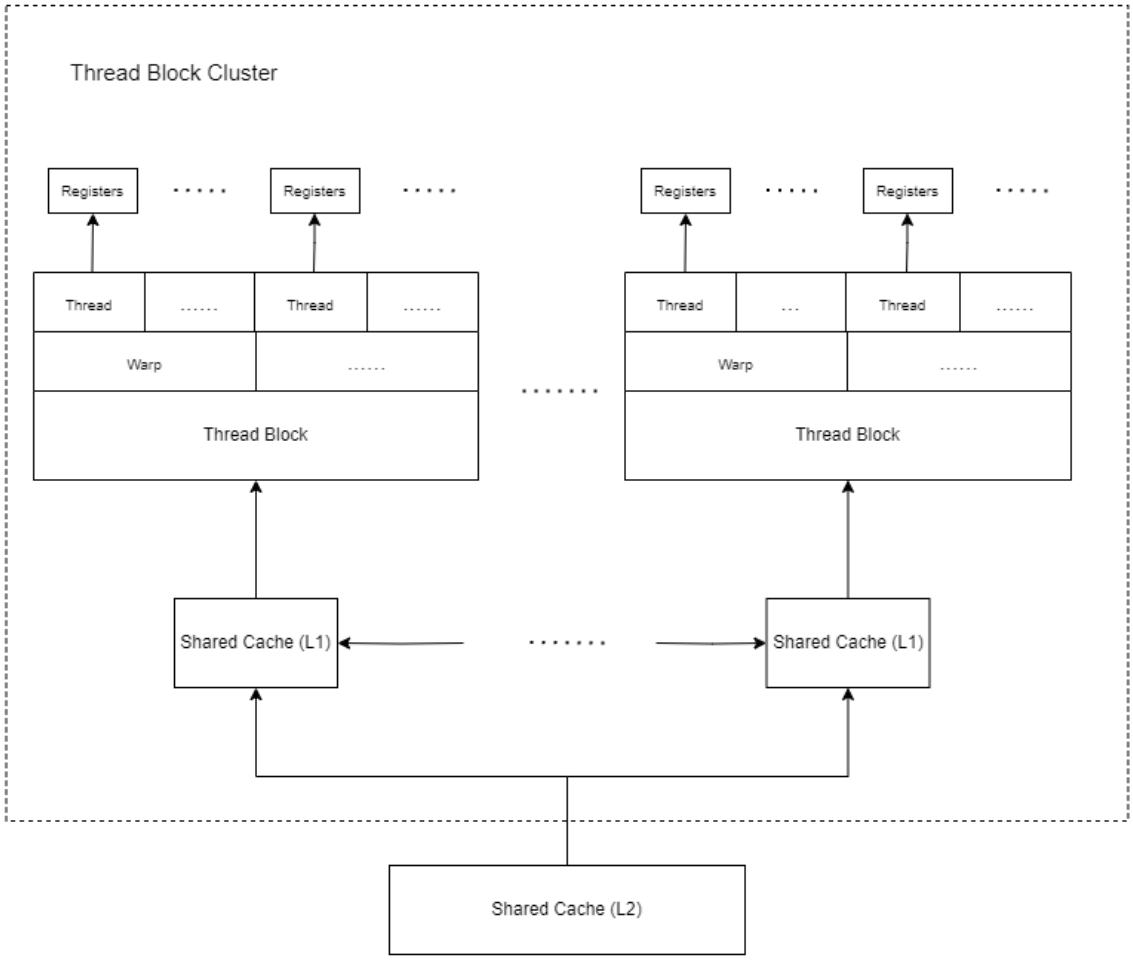

Figure 2.1: an abstract overview of data transfer within a thread block cluster.

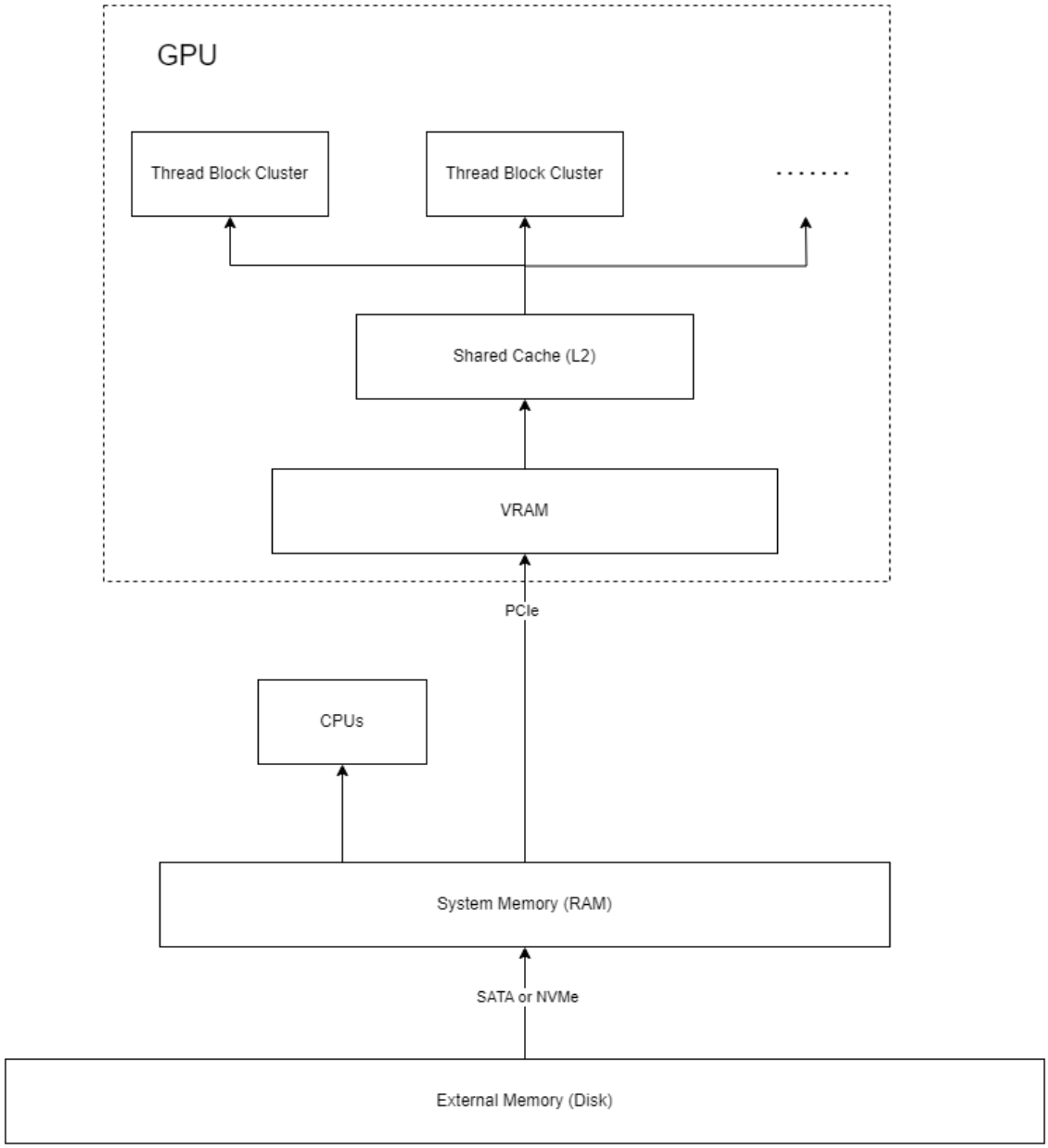

Figure 2.2: an abstract overview of data transfer from external memory to the GPU.

At the top level, the GPU comprises multiple thread block clusters, each containing several thread blocks that are guaranteed to be concurrently scheduled onto a group of SMs [10]. Each thread block has several warps defined by the programmers. Each warp consists of 32 threads and each thread has a dedicated register. The threads in a warp are executed concurrently in the SIMD (Single Instruction, Multiple Data)

fashion.

Each thread block has access to similar-sized shared memory and L1 cache. These L1 caches feed into a larger shared cache (L2) which stores temporal data for efficient retrieval by the GPU. The L2 cache connects to the GPU's VRAM, which interfaces with the system's main memory (RAM) through the PCIe bus. The CPU also interacts with the system memory linked to external storage (disk) via SATA or NVMe connections.

Here are some key features to note about data transfer:

1. Data contention occurs in both CPU and GPU systems, as cache conflicts in the CPU and bank conflicts in the GPU.

2. In most architectures, the smallest unit of data transfer is a block of consecutive data, which can vary in size. The memory controller can combine multiple memory accesses of a block into a single transaction. Therefore, accessing contiguous memory from a group of threads, when possible, greatly improves the efficiency of data transfer.

3. The GPU shared cache (L2) is too small for temporal locality, as the number of threads is too large for this cache to handle. As a result, the L2 is often deprecated, especially since the total size of the private caches (L1) is comparable to that of the L2 cache.

4. Data transfer latency significantly increases with deeper hierarchical storage levels. For example, PCIe is much faster than SATA or NVMe.

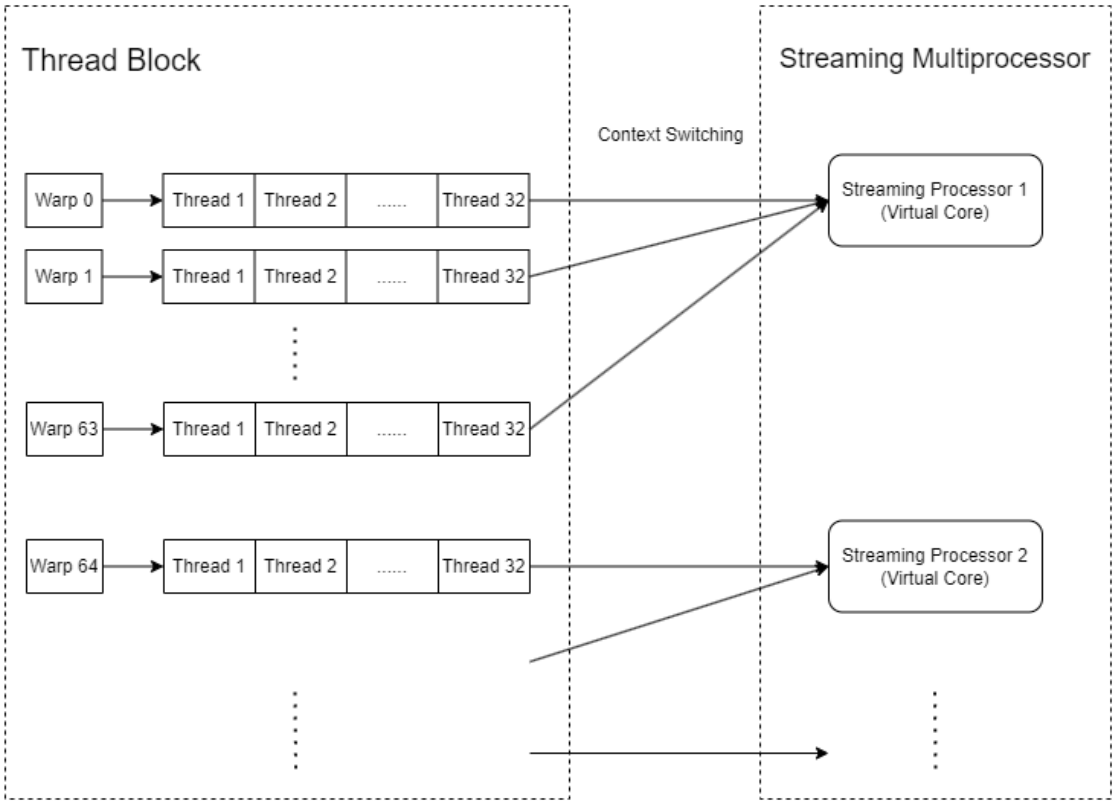

Figure 2.3: an abstract overview of GPU computation architecture showcases the context switching of a fixed number of warps into and out of one assigned Streaming processor.

Here are some key features to note about computation:

1. In core groups, context switching for each processor is managed by a warp. Typically, a warp consists of 32 threads that execute each single instruction simultaneously. Consequently, branch instructions that cause branch divergence potentially require additional time to account for all possible divergent paths.

2. GPUs can do much faster context switching than CPUs because of the sizeable register file. So if enough parallelism and computations occur, the latency of thread operations and sometimes data transfer are implicitly hidden. Remark that these are the opposite for CPUs. People try to reduce latency by techniques (e.g. branch prediction, out-of-order execution, pipelining). Furthermore, CPU threads are blocked when they do data operations.

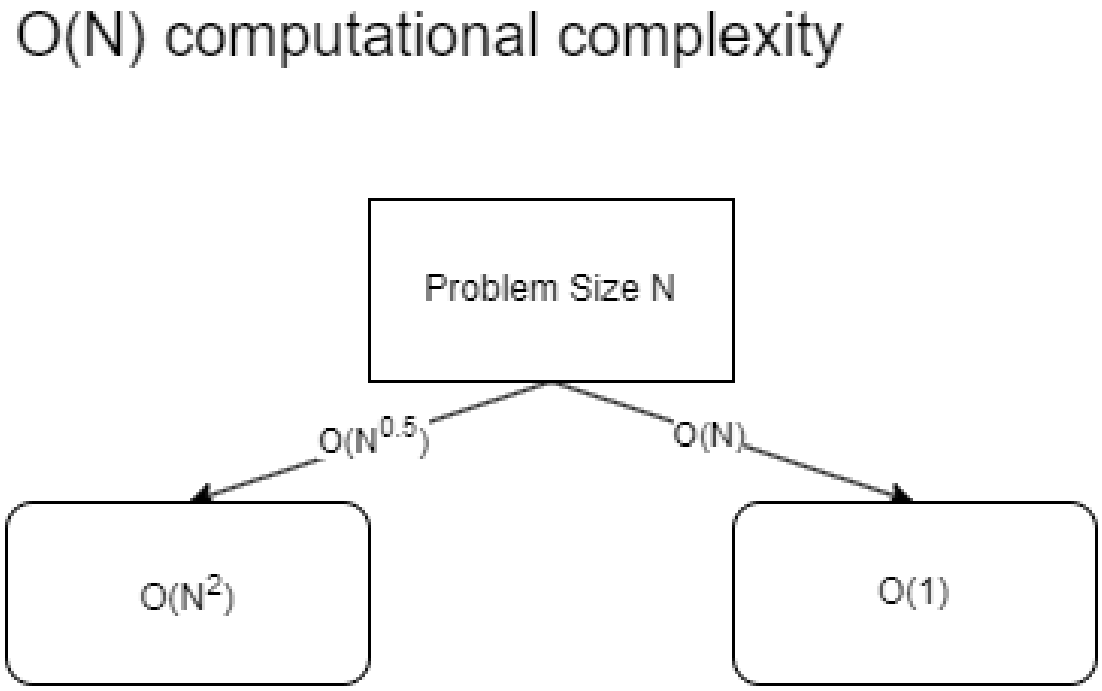

Figure 2.4: an example of branch divergence.

One can see that models that give asymptotic analysis cannot model all of the above features, such as branch divergence in computation feature 1. In Fig 2.4, one can argue a trivial computational complexity analysis of branches is O(maximum of computations of branches), which is O(N^2) without knowing the size of data flows. If given all threads in each warp run in a single branch, such that O($\sqrt{N}$) of data flows into the O(N^2) complexity branch and the rest flows to the O(1) complexity branch, one can also show the complexity is O(N). Such features are not responsible for the models to handle; instead, one should provide specific proof for the behaviour of algorithms.

In another case, if both branches have the same computational complexity, the asymptotic analysis would be the same, regardless of the occurrences of branch divergences.

Chapter 3

Model of Analysis

Table 3.1: GPU Hardware

Parameter	Description
N_{SM}	number of utilizable SMs
P	number of SPs per SM
M_{SM}	size of shared Memory per SM
G	size of global Memory

Table 3.2: GPU Computation

Parameter	Description
K	number of kernel launch
N_T	number of thread launch
H_i	highest number of computation operations passed by any thread at ith kernel
W_i	total number of computation operations executed by all threads at ith kernel

Firstly, we define the basic parameters for the model, while one can introduce more in their proofs.

Model Assumptions:

The computation and data operation unit is a word. The problem size is arbitrarily large with N words. The model can then assume that all threads in one streaming

multiprocessor (SM) run in the SIMD manner. The computation is scheduled optimally at no cost by context switching since no hardware scheduling protocol (cache coherency, preemption etc.) is assumed in this model for flexibility and simplicity.

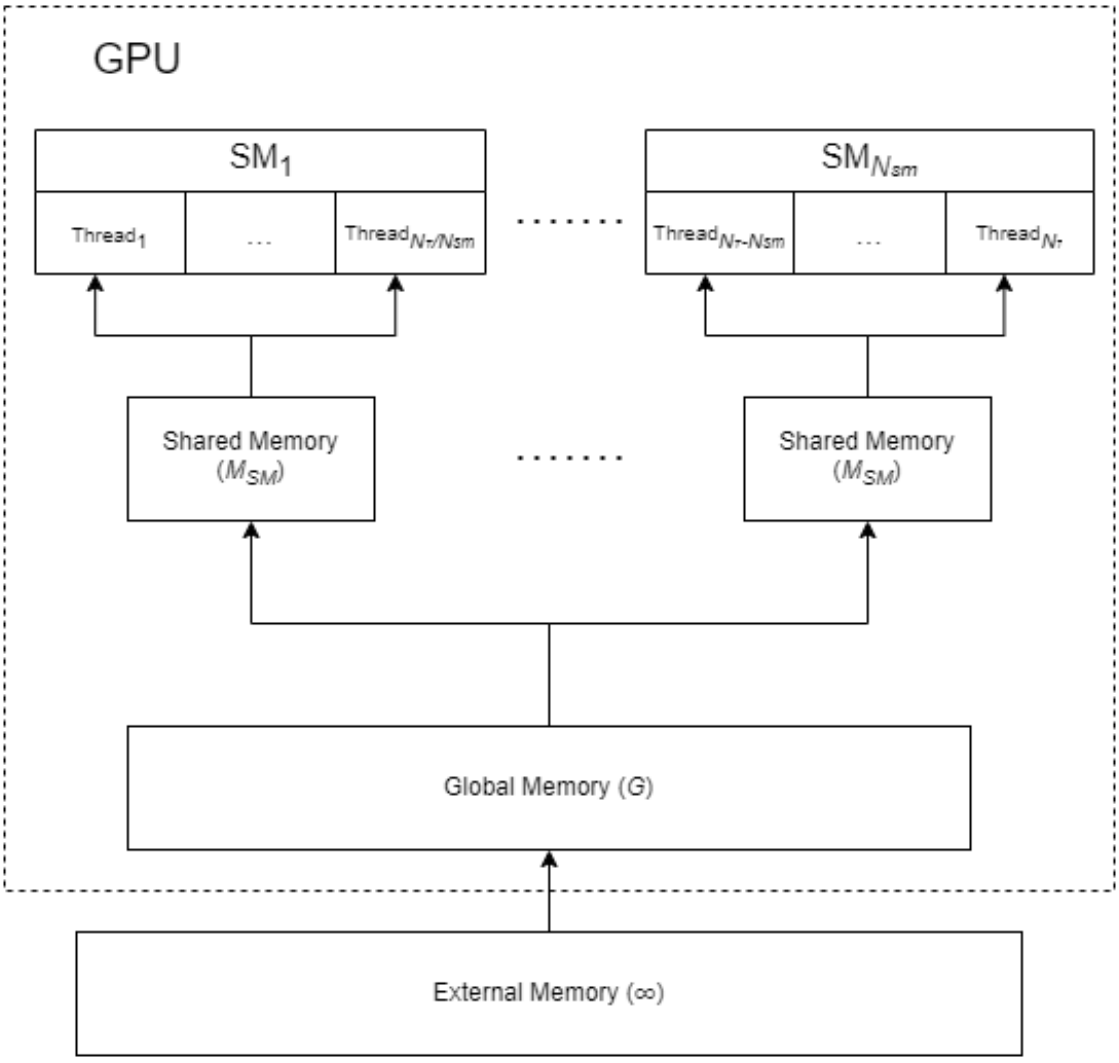

Figure 3.1: overview of the model.

Model Setup:

1. The problem with a predefined data structure is initially stored in either the infinite-sized external memory or global memory. One should give the assumption of the size of global memory G when necessary.

2. One block of B words transfer between shared and external memory requires O(1) slow memory operations.

3. One sequence of word transfer between global and external memory requires O(1) fast memory operations.

4. Each processor takes O(1) operations to access data in its cache

The model evaluates the performance by a tuple of metrics to capture the bottlenecks of algorithms:

1. Number of slow memory operations performed in the algorithm

2. Number of fast memory operations performed in the algorithm

3. K

4. $\sum_{i=0}^{K} H_i$

5. $\sum_{i=0}^{K} W_i$

To avoid confusion in metrics 4 and 5. Let me showcase a simple computation analysis of a kernel launch i:

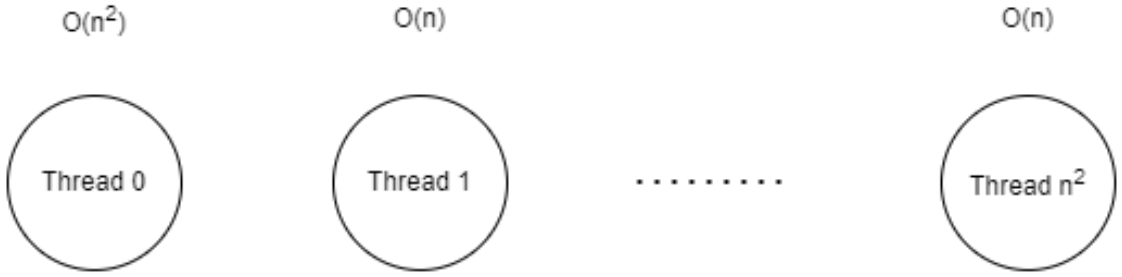

Figure 3.2: computation analysis example.

In 'Figure 3.2', we have one thread 'Thread 0' assigned to $O(n^2)$ computation operations, and the rest of n^2 threads are assigned to $O(n)$ computation operations. In this case, 'Thread 0' performs the highest number of computation operations in the ith kernel, resulting in $H_i = O(n^2)$. One thread does $O(n^2)$ computation operations, and the rest of the threads do $n^2 * O(n)$ total computation operations, resulting $W_i = O(n^3)$

Chapter 4

Model Applicability

In this section, I will explain design thinking and trade-offs in this model so algorithm designers can determine if the model applies to their algorithm analysis.

We overlooked operation-level parallelism, as other well-known models did because it will introduce numerous parameters, such as the size of each type of execution unit. In addition, it is too detailed for algorithm designers to analyze the instruction-level operations.

We integrated the L1 cache with shared memory. The shared memory can be explicitly managed, whereas the L1 cache operates as passive storage. Both have comparable sizes and occupy the same hierarchical level in our model. Even if algorithm designers do not explicitly utilize shared memory, they can still analyze the implicit use of the L1 cache.

The two-level memory access hierarchy exhibits a few unique characteristics in our model. They aim to capture the unique memory access pattern of numerous threads that leverage the extensive available VRAM bandwidth. This design also incorporates the benefits of coalesced memory access and elegantly captures the possible issue of bank conflicts. This design also allows algorithm designers to parameterize the number of I/O ports available for each bank when possible.

The metrics 4 and 5 indicate the bottleneck of computations:

$$\sum_{i=0}^{K} H_i$$

provides the estimation of computation time, assuming that even the heaviest threads are always executed in streaming processors.

However, the computation time is also limited by the amount of work completed relative to the number of available processing units.

$$\sum_{i=0}^{K} \frac{W_i}{N_{SM} * P}$$

estimate the computation time, assuming that even the works are scheduled perfectly for each streaming processor.

Chapter 5

Algorithm Design

The model we just developed allows us to formalize and evaluate much more complicated algorithms more abstractly. In this section, we aimed to present a parallelized dynamic graph algorithm to demonstrate the model's application. However, many well-known sequential data structures have yet to be adapted for parallel or GPU settings.

5.1 Problem setup:

The problem we tried to solve is called decremental reachability in a directed acyclic graph. Given a directed acyclic graph $G_0 := (V, E_0)$ with standard notations $n := |V|$, $m := |E_0|$; $V := \{v_0, v_1, ..., v_{n-1}\}$ where $s := v_0$; $E \subseteq V \times V$; sequences of edge deletions $(D_i)_{i=1}^{d}$. After each sequence D_i, one should be able to answer if a vertex is reachable from s in graph $G_i := (V, E_{i-1} \backslash D_i)$ within O(1) computation and memory operations.

One can solve this problem using a reachability algorithm such as breadth-first search in a static setting. However, running these algorithms after each deletion is quite costly. Since each deletion typically does not significantly alter the graph, we can design more efficient data structures and algorithms to amortize the cost over the entire process and handle multiple deletions in parallel. And minimize the query time.

This is a classic problem in the theoretical computer science area. While researchers

typically focus on optimizing serialized algorithms, recent breakthroughs[3] [9] [6] often utilize the even and shiloach tree, another classic data structure which is generally considered difficult to parallelize. Here, we do not propose the same approach but rather provide a starting point for parallelizing such algorithms in the GPU system without extensive optimization efforts.

It is evident that a vertex remains reachable if and only if s continues to be an ancestor following edge deletions. So one can keep track of the indegrees of ancestors to derive algorithms. We give two different design styles of the algorithm with this essence.

Without loss of generality, we assume that all vertex is reachable from s from the start. Global memory has $O(n)$ space, which is sensible since more than billions of vertices are not for a single system. Concurrent read concurrent write (CRCW) is assumed for the read/write conflict strategy. We run the "Host" for each sequence of deletion queries.

5.2 Vertex-Centric Algorithm:

Algorithm 1 Host 1

Input (stored in external memory):
    A: adjacency matrix stored in row span
    $D := D_i$: a sequence of edge deletion queries
Initialize (stored in global memory):
    Q_0, Q_1: compact buffer with size counter to swap in global memory
    L: array where $L[i]$ stores indegree of vertex i
    $L[0] \leftarrow 1$; $Q \leftarrow Q_0$
run **Query**
while Q is not empty **do**
    run **Propagation**
    size counter of $Q \leftarrow 0$
    swap Q with another buffer
end while

Algorithm 2 Query

Launch $|D|$ threads
each thread with global ID i:
attemptDeletion($D[i]$)

Algorithm 3 Propagation

1: Launch $|Q| * n$ threads
2: each thread with global ID i:
3: attemptDeletion($(Q[i//n], i\%n)$)

Algorithm 4 attemptDeletion

```
1: function attemptDeletion(edge (u, v))
2:     if atomicCAS(A[u, v], 1, 0) = 1 then
3:         if atomicAdd(L[v], -1) = 1 then
4:             enqueue v to another buffer than current Q
5:         end if
6:     end if
7: end function
```

Comment of *attemptDeletion*: When an edge is pending to be deleted, atomically check if A[u, v] is 1; if it is, delete it; then atomically decrease the indegree of v by 1 in L[v], if it gets the point where v has 0 indegree, then put it to another buffer to be ready to delete all its outedges in next kernel.

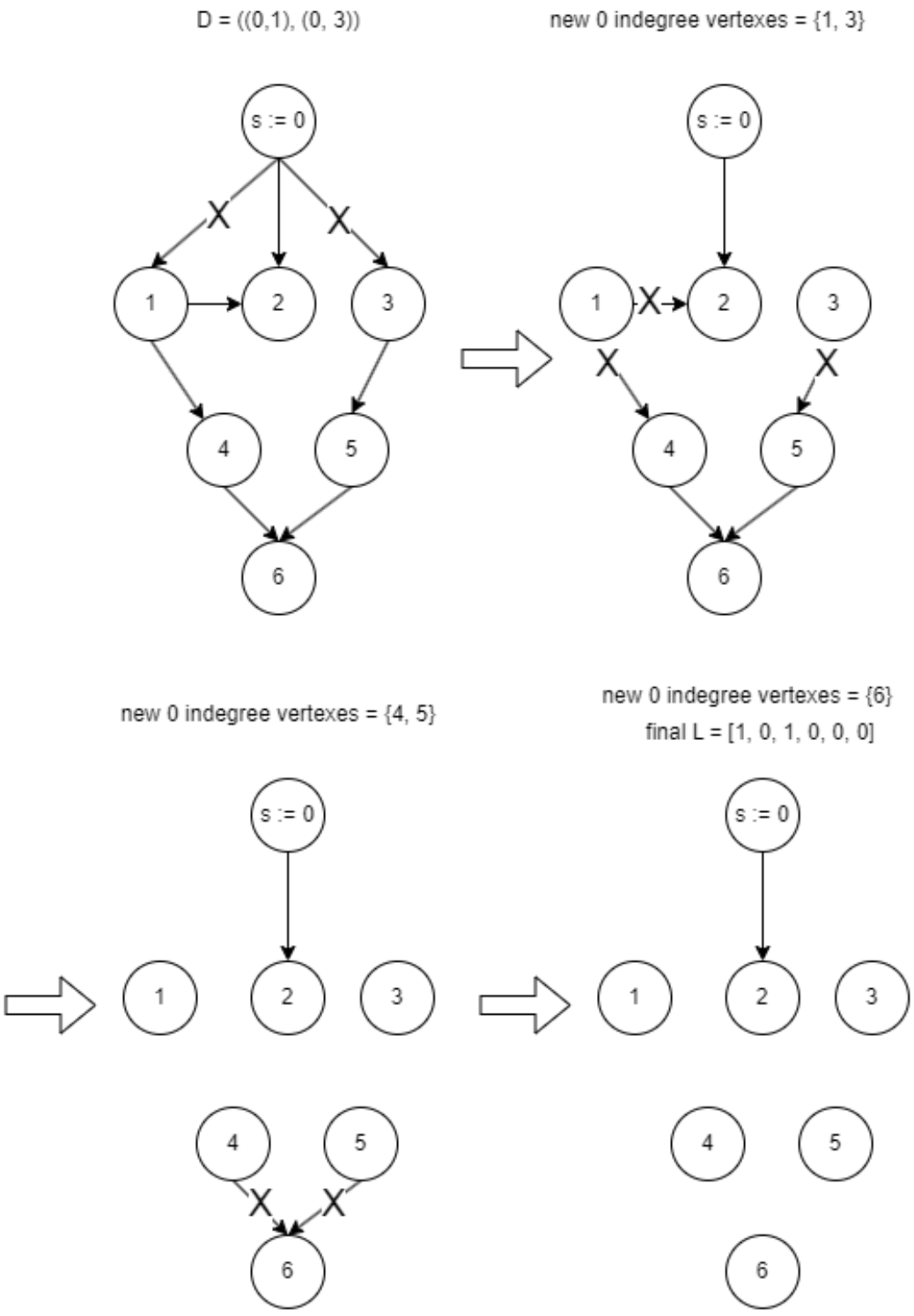

Figure 5.1: simplified example of both 5.2 & 5.3 algorithms running

5.2.1 Analysis:

We analyze the entire process of the algorithm until all vertices are no longer reachable from s:

1. Number of slow memory operations:

We access the external memory in "Host 1" line 3 when we read queries and "attemptDeletion" line 2.

"Query" randomly access A in external memory $O(\sum_{i=1}^{d} |D_i|)$ times by "attemptDeletion".

When a vertex is no longer reachable(has 0 indegree), deleting its outedges induces $O(\frac{n}{B})$ operations to access the corresponding row in A. Observe that this operation is taken once per vertex in the entire process. The total operations is $O(\frac{n^2}{B})$. Overall the total is $O(\frac{n^2}{B} + \sum_{i=1}^{d} |D_i|)$

2. Number of fast memory operations:

There is no guaranteed access pattern for fast memory operations which induces $O(m + \sum_{i=1}^{d} |D_i|)$ in "attemptDeletion" lines 2 to 4.

3. K:

The worst case is $m + d$ when "Query" or "Propagation" deletes only one edge at a time in each kernel.

4. $\sum_{i=0}^{K} H_i$:

In the worst case, all threads can wait for $O(m)$ operations to enqueue their vertex over the entire process.

5. $\sum_{i=0}^{K} W_i$:

The total computation operations done by all threads are $O(n^2)$ for "Propagation" and $O(\sum_{i=1}^{d} |D_i|)$ for "Query" over entire process.

So overall $O(n^2 + \sum_{i=1}^{d} |D_i|)$

The vertex-centric algorithm described above exemplifies a common design in CPU

systems, where the primary focus is on the number of computations. However, this approach can result in a low occupancy rate, leading to significant overhead from kernel launches in GPU systems, which is a critical metric in our model.

5.3 Edge-Centric Algorithm:

Now, we have developed a concise edge-centric algorithm that is more GPU-friendly. This will illustrate the crucial trade-offs in designing GPU algorithms within our model.

Algorithm 5 Host 2

Input (stored in external memory):
    A: transpose of adjacency matrix stored in row span
    $D := D_i$: a sequence of edge deletion queries
Initialize (stored in global memory):
    I: boolean indicates whether there is new non-indegree vertex
    $L1, L2$: indicates whether vertex i is reachable
    $I \leftarrow 1$; $\forall i, L_1[i] \leftarrow 1, L_2[i] \leftarrow 0$
run **EdgeCheck**

Algorithm 6 EdgeCheck

Launch n^2 threads
each thread with global ID i:
for $j \leftarrow 0$ to $\left(\frac{|D|-i}{n^2}\right)$ **do** ▷ boundary limit $|D|$
    $spos \leftarrow i + j \cdot n^2$ ▷ current sequence position
    $A[D[spos][0], D[spos][1]] \leftarrow 0$
end for
grid.sync()
while I **do**
    if $i = 0$ **then** ▷ One thread turns I off
        $I \leftarrow 0$
    end if
    grid.sync()
    if $L_1[\frac{i}{n}]\&A\left[\frac{i}{n}, i\%n\right]$ **then**
        $L_2[\frac{i}{n}] \leftarrow 1$
    end if
    grid.sync()
    if $i < n$ **then**
        $x \leftarrow L_2[i]$
        $L_2[i] \leftarrow 0$ ▷ initialization of temporary array
        if $!x\&L_1[i]$ **then**
            $L_1[i] \leftarrow 0$
            $I \leftarrow 1$ ▷ need an extra iteration
        end if
    end if
    grid.sync()
end while

5.3.1 Analysis:

A similar style of analysis applies in the algorithm "EdgeCheck".

1. Number of slow memory operations:

In line 5, A is accessed once for each deletion query, so overall $O(\sum_{i=1}^{d} |D_i|)$ operations. Line 14 is accessed once for all threads at the start of each kernel and after each vertex becomes a 0-indegree vertex in the worst case. But they all coalesced accesses by rows of A. So overall $O(\frac{n^2 \cdot (n+d)}{B})$ operations in the worst case.

2. Number of fast memory operations:

In line 5, even though accesses of D are coalesced, it is not for A. So these take $O(\frac{\sum_{i=1}^{d} |D_i|}{N_{SM}})$ in worst case. One can easily see that accesses of A line 14 are dominant in the "while loop". Although they are coalesced, they still take $O(\frac{n^2}{N_{SM}})$ each time. So overall $O(\frac{n^2 \cdot (n+d)}{N_{SM}})$ operations in the worst case.

3. K:

This version of the algorithm trades off the number of kernel launches against other metrics. One can easily see the number of kernel launches is $max(m, d)$ for the entire process, which is way more efficient than the Vertex-Centric Algorithm.

4. $\sum_{i=0}^{K} H_i$:

The maximum number of operations that one thread does is $O(\sum_{i=1}^{d} \frac{|D_i|}{n^2})$ in the "for loop". Constant operations are done in the "while loop" for each thread. So overall $O(d + n)$ operations.

5. $\sum_{i=0}^{K} W_i$:

The total number of operations done for all threads is $O(\sum_{i=1}^{d} |D_i|)$ in the "for loop". The searching for the entire matrix A dominates the number of operations in the "while loop", which is $O(n^2 \cdot (n + d))$ operations.

Table 5.1: Algorithm comparisons

	Vertex	Edge
slow memory	$O(\frac{n^2}{B} + \sum_{i=1}^{d} \lvert D_i \rvert)$	$O(\frac{n^2 \cdot (n+d)}{B})$
fast memory	$O(m + \sum_{i=1}^{d} \lvert D_i \rvert)$	$O(\frac{n^2 \cdot (n+d)}{N_{SM}})$
K	$m + d$	$max(m, d)$
$\sum_{i=0}^{K} H_i$	$O(m)$	$O(d + n)$
$\sum_{i=0}^{K} W_i$	$O(n^2)$	$O(n^2 \cdot (n + d))$

When algorithm designers develop the algorithms and analyses outlined in Table 5.1, system researchers can start to select the appropriate algorithm for deployment within their systems. Here are a few examples:

1. Broadly, if the system, GPU may be outdated compared to the memory system, one should consider pivoting more to algorithms with lower computational complexity, and vice versa.

2. In some cases, the problem involves extensive computations, but the data size is small enough to fit in VRAM. In such instances, it may be advisable to disregard the metric 'slow memory operation'.

3. GPU kernel launches inherently cause latency, particularly noticeable when the computation within the kernel is light(e.g. our vertex-centric algorithm). If an algorithm requires frequent kernel launches and your CPU and GPU are somewhat outdated, it may be wise to shift towards computationally less efficient algorithms that offer better parallelism.

Chapter 6

Conclusion and Future Work

In this industrial project, we provided an overview of key features of the latest NVIDIA GPU architecture. Subsequently, we designed a model to analyze massively parallelized algorithms in GPU systems. We developed two dynamic graph algorithms with different designs, making trade-offs between specific metrics. These algorithms were analyzed and compared to demonstrate the application of our model and its capabilities. Experimental results on different GPUs would be helpful here. But hopefully, I will have the budget for that in future research.

The GPU architecture is slightly changed over time. For example, the hierarchical layer "thread block cluster" was recently introduced in the latest H100 Tensor Core GPU. When the model becomes "outdated" due to rare significant architecture adjustments, one should adjust their analysis or formalize a modified model accordingly. However, this project provides a solid baseline for analyzing such algorithms. So, the main ideas should be similar.

Another concern is that GPU algorithm researchers do not have many provably efficient parallelized data structures available when designing algorithms. This is especially true for graph representation data structures, as we encountered challenges during the design of the two algorithms. If my research continues in this area, I will attempt to address fundamental problems, such as parallelizing basic data structures or redefining them to adapt to GPUs.

Chapter 7

Glossary of Terms

Terms	Description
Branch Divergence	Occurs when threads within the same warp follow different execution paths due to conditional statements, leading to inefficient use of the GPU's parallel processing capabilities.
Computation Operation	An operation that performs arithmetic or logical calculations on data in GPU's processing cores.
Data Operation	An operation involves transferring, loading, or storing data in memory, such as reading or writing to global or shared memory.
Kernel	A function executed on the GPU by many parallel threads initiated by a CPU.
Coalesced Memory Access	A memory access pattern where consecutive threads in a warp access consecutive memory locations, allowing the GPU to combine these accesses into a single memory transaction to improve efficiency.
Decremental Graph Algorithm	Algorithm that focuses on efficiently updating the solution when edges or nodes are removed from the graph, as opposed to recalculating the solution from scratch.
Atomic Operations	Low-level operations guarantee exclusive access to a memory location, ensuring that a variable is updated correctly even when multiple threads attempt to modify it simultaneously.

Bibliography

[1] J.A. Ang, R.F. Barrett, R.E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S.D. Hammond, K.S. Hemmert, S.M. Kelly, H. Le, V.J. Leung, D.R. Resnick, A.F. Rodrigues, J. Shalf, D. Stark, D. Unat, and N.J. Wright. Abstract machine models and proxy architectures for exascale computing. In *2014 Hardware-Software Co-Design for High Performance Computing*, pages 25–32, 2014.

[2] Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, page 197–206, New York, NY, USA, 2008. Association for Computing Machinery.

[3] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental reachability, scc, and shortest paths via directed expanders and congestion balancing. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1123–1134, 2020.

[4] Ulrich Drepper. What every programmer should know about memory. http://www.akkadia.org/drepper/cpumemory.pdf, 2007. Red Hat, Inc.

[5] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, page 114–118, New York, NY, USA, 1978. Association for Computing Machinery.

[6] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs. In *Proceedings of the Forty-Sixth Annual ACM Symposium on*

Theory of Computing, STOC '14, page 674–683, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Adam Jacobs. The pathologies of big data: Scale up your datasets enough and all your apps will come undone. what are the typical problems and where do the bottlenecks generally surface? *Queue*, 7(6):10–19, jul 2009.

[8] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. A learned performance model for tensor processing units. volume 3, pages 387–400, 2021.

[9] Jakub Lacki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Algorithms*, 9(3), jun 2013.

[10] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. Benchmarking and dissecting the nvidia hopper gpu architecture, 2024.

[11] John McCarthy. A basis for a mathematical theory of computation, preliminary report. In *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, pages 225–238, 1961.

[12] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[13] Sergio Rivas-Gomez, Antonio J Pena, David Moloney, Erwin Laure, and Stefano Markidis. Exploring the vision processing unit as co-processor for inference. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 589–598. IEEE, 2018.