Data-Centric Computing Techniques for Out-of-Core GPU Applications

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2022

Jingcheng Shen

Published Papers

Journal Papers

1. J. Shen, K. Shigeoka, F. Ino, and K. Hagihara. GPU-based branch-and-bound method to solve large 0-1 knapsack problems with data-centric strategies. *Concurrency and Computation: Practice and Experience*, 31(4):e4954, 2019.

2. J. Shen, F. Ino, A. Farrés, and M. Hanzich. A data-centric directive-based framework to accelerate out-of-core stencil computation on a GPU. *IEICE Transactions on Information and Systems*, E103-D(12):2421–2434, 2020.

International Conference Papers

1. J. Shen, K. Shigeoka, F. Ino, and K. Hagihara. An out-of-core branch and bound method for solving the 0-1 knapsack problem on a GPU. In *Proceedings of the 17th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 254–267, 2017.

2. R. Zhu, J. Shen, X. Deng, M. Walldén, and F. Ino. Training strategies for CNN-based models to parse complex floor plans. In *Proceedings of the 9th International Conference on Software and Computer Applications (ICSCA)*, pages 11–16, 2020.

3. J. Shen, C. Fu, X. Deng, and F. Ino. A study on training story generation models based on event representations. In *Proceedings of the 3rd International Conference on Artificial Intelligence and Big Data (ICAIBD)*, pages 210–214, 2020.

4. J. Shen, J. Mei, M. Walldén, and F. Ino. Integrating GPU support for FreeSurfer with OpenACC. In *Proceedings of the 6th IEEE International Conference on Computer and Communications (ICCC)*, pages 1622–1628, 2020.

5. Y. Wu, J. Shen, M. Okita, and F. Ino. Accelerating a lossy compression method with fine-grained parallelism on a GPU. In *Proceedings of the 12th International Symposium*

on Parallel Architectures, Algorithms and Programming (PAAP), accepted, 2021.

6. J. Shen, Y. Wu, M. Okita, and F. Ino. Accelerating GPU-based out-of-core stencil computation with on-the-fly compression. In *Proceedings of the 22nd International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, accepted, 2021.

Oral Presentations

1. J. Shen, K. Shigeoka, F. Ino, and K. Hagihara. An out-of-core CPU-GPU cooperative B&B solver for the large knapsack problem. *Poster in the 2nd Cross-disciplinary Workshop on Computing Systems, Infrastructures, and Programming (xSIG)*, 2018.

2. J. Shen, N. Miki, F. Ino, K. Hagihara, A. Farrés, and M. Hanzich. Applying PACC directives to accelerate out-of-core seismic wave simulation on a GPU. *PhD Forum in the 33rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2019.

3. J. Shen, N. Miki, and F. Ino. Evolving PACC framework with data-centric optimizations for out-of-core stencil computation. *Invited Talk in User Feedback session in the OpenACC Annual Meeting (OpenACC)*, 2019.

4. J. Shen, F. Ino, A. Farrés, and M. Hanzich. Accelerating large seismic simulation code with PACC framework. *Poster in the 11th GPU Technology Conference (GTC)*, 2020.

Abstract

Graphics processing units (GPUs) are highly efficient devices for high-performance computing (HPC) and are widely adopted to construct heterogeneous (i.e., CPU-GPU) supercomputers. However, the small GPU memory capacity necessitates the use of out-of-core computation to process excess data, involving the time-consuming CPU-GPU data transfer prone to limit the overall performance.

In this thesis, for two out-of-core GPU applications, we propose data-centric computing techniques to address the problem that the CPU-GPU data transfer limits the overall performance. The first application is a branch-and-bound (B&B) method to solve the 0-1 knapsack problem, which is a widely used combinatorial optimizer. The second application is stencil computation that occurs in various scientific fields. The proposed techniques significantly reduce the CPU-GPU data transfer, improving the performance of the studied applications. Moreover, the two applications have very different runtime characteristics. Most notably, the B&B 0-1 knapsack solver changes memory footprint at runtime, whereas the stencil computation uses a fixed amount of memory at runtime. Despite the difference between applications, the proposed techniques follow the same mindset of data-centric computing: making as much use as possible of the data residing on the GPU and reducing as much as possible of the CPU-GPU data transfer.

The proposed data-centric computing techniques are in detail explained in three parts of the thesis. In the first part, an out-of-core B&B method to solve large 0-1 knapsack problems on a GPU is proposed. Given a large problem that produces many subproblems, the proposed method dynamically swaps subproblems to CPU memory. Because such a CPU-centric subproblem management scheme increases CPU-GPU data transfer, we adopt three data-centric techniques to eliminate this side effect, including (1) an out-of-order search (O3S) technique that reduces the data transfer overhead by adaptively transferring subproblems between the CPU and GPU, (2) a GPU stream compaction technique that reduces the sparseness of arrays to be transferred, and (3) an explicitly-managed pipelining technique that hides the data transfer overhead by overlapping data transfer with GPU B&B operations. Experimental results demonstrate that the out-of-core solver with proposed techniques stored 41× as

many subproblems as a previous in-core solver that manages subproblems in GPU memory, solving approximately twice as many problem instances on the GPU. In addition, compared to a previous breadth-first search (BFS) technique, the proposed O3S technique achieved an average speedup of 7.5×.

In the second part, we extend a stencil framework to address data transfer problems. We propose two data-centric computing techniques: (1) a direct-mapping technique that eliminates host (i.e., CPU) buffers, which intermediate between the original data and device buffers, and (2) a region-sharing technique that significantly reduces CPU-GPU data transfer by allowing contiguous data chunks to share common regions on the GPU. The extended framework was applied to an acoustic wave propagator, automatically extending the length of the original serial code 2.3-fold to generate the out-of-core code. Experimental results reveal that the generated code ran 41.0×, 22.1×, and 3.6× as fast as implementations based on Open Multi-Processing (OpenMP), Unified Memory, and the previous framework, respectively. The generated code also demonstrates usefulness with small datasets that fit in the device capacity, running 1.3× as fast as an in-core implementation.

In the third part, we propose a data-centric technique that further accelerates out-of-core GPU stencil computation with on-the-fly GPU compression, introducing a novel data compression scheme that solves the data dependency between contiguous decomposed data chunks. We also modify a widely used GPU compression library to support pipelining that overlaps data transfer with GPU computation. Experimental results show that the stencil code with the proposed technique achieved a speedup of 1.13× with a tolerable fidelity loss, compared with a code that involves no compression.

Our work demonstrates that data transfer should be weighed more than computation in optimizing large GPU codes. The two studied applications show that the proposed data-centric computing techniques effectively improve overall performance and thus emphasize the importance of data-centric computing techniques to present and future large-scale GPU applications.

Acknowledgements

Foremost, I would like to express my deepest gratitude to my supervisor, Professor Fumihiko Ino, for his continuous support of my Ph.D. research and daily life during my years. His valuable and insightful guidance has helped me in fulfilling the requirements for graduation including this thesis.

Besides my supervisor, I would like to thank the rest of my thesis committee: Professor Katsuro Inoue and Professor Toshimitsu Masuzawa for their insightful comments and encouragement.

I am deeply grateful to Associate Professor Masao Okita and Assistant Professor Koki Masui for discussions with them, for their many valuable and thoughtful comments, and for their encouragement.

I would like to thank my collaborators: Dr. Albert Farres, Dr. Mauricio Hanzich (affiliated with the Barcelona Supercomputing Center), and Dr. Jie Mei (affiliated with the Department of Computer Science & Brain and Mind Institute, Western University). Dr. Albert Farres and Dr. Mauricio Hanzich offered me an internship to collaborate with their research group and gave me a lot of motivating research ideas and guided me through the difficulties during the research. Dr. Jie Mei gave me ideas for interdisciplinary research.

I would like to express my sincerest gratitude to Humanware Innovation Program at Osaka University, Student Researcher Program offered by Daikin Industries, Inc., and the Graduate School of Information Science and Technology at Osaka University, which successively provided me with financial support.

Finally, I am deeply indebted to members of the Ino laboratory, for their daily support and useful comments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we first brief the history of high-performance computing (HPC). We then describe the recent heterogeneous computing architecture with graphics processing units (GPUs) and the bottleneck for accelerating applications on this architecture. Finally, we summarize the contributions of this thesis.

1.1 Overview of HPC

HPC, with many synonyms such as supercomputing and massively parallel computing, is one of the most important research fields in computer science. HPC machines, exemplified by supercomputers, excel in handling a wide range of data- and compute-intensive tasks in various fields such as physical simulations, climate science, molecular modeling, and so forth. The key to HPC machines' success is a large number of floating-point operations per second (FLOPS), which must be secured by both HPC hardware and software. In addition to hardware components such as massive compute units, high-bandwidth memory, and high-throughput networks, software factors such as well-optimized parallel codes and efficient system software to manage resources and tasks are inevitable to yield a high FLOPS.

UNIVAC LARC, built in 1960, is considered among the first supercomputers. Released in 1972, ILLIAC IV was the first true *massively parallel* computer. Figure 1.1 shows the outstanding supercomputers at the early days of each decade from 1960 to today, demonstrating the performance of supercomputers is increasing exponentially over time [101, 102]. Japan and the United States are two leading players in the arms race of supercomputing, with China as the strongest challenger known for its Sunway TaihuLight [22].

Entering the era of exascale computing (10^{18} FLOPS a.k.a 1 ExaFLOPS), GPU, as the very representative of accelerators, is gaining momentum for the great capability of massively parallel computing, compared to a CPU. Besides the traditional homogeneous architecture that only utilizes CPUs, the heterogeneous architecture becomes a promising

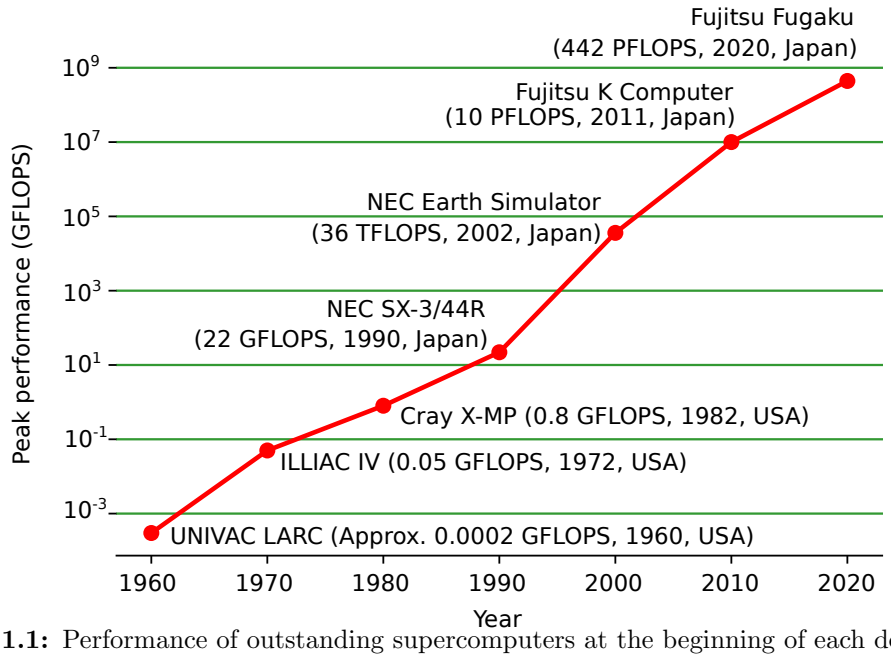

Figure 1.1: Performance of outstanding supercomputers at the beginning of each decade from 1960.

Table 1.1: Comparison of contemporary high-performance CPU and GPU. Note that "SP" and "DP" represent "single-precision" and "double-precision," respectively.

	IBM POWER9 CPU	NVIDIA Tesla V100 GPU
Release date	2017	2017
Number of compute cores	22	5120
Peak performance (SP/DP, TFLOPS)	1.1/0.5	15.7/7.9
Memory bandwidth (GB/s)	170	900

option that uses both CPUs and GPUs to construct supercomputers. In recent years, CPU-GPU supercomputers keep leading the *TOP500* supercomputer ranking. For instance, the CPU-GPU supercomputer Summit [79] had been the fastest in the list from June 2018 to November 2019. Currently, the CPU-GPU supercomputers Summit and Sierra [40] rank the second and third in the list, respectively. Table 1.1 compares the CPU and its counterpart GPU used in the Summit and Sierra supercomputers. With respect to FLOPS per unit, a GPU is beyond 10× as good as a CPU, which reduces the necessary number of compute units to construct a supercomputer by a factor of ten. Moreover, a GPU is very power-efficient compared to a CPU. Summarily, the CPU-GPU architecture is an excellent solution for supercomputing in terms of reducing constructing and running costs.

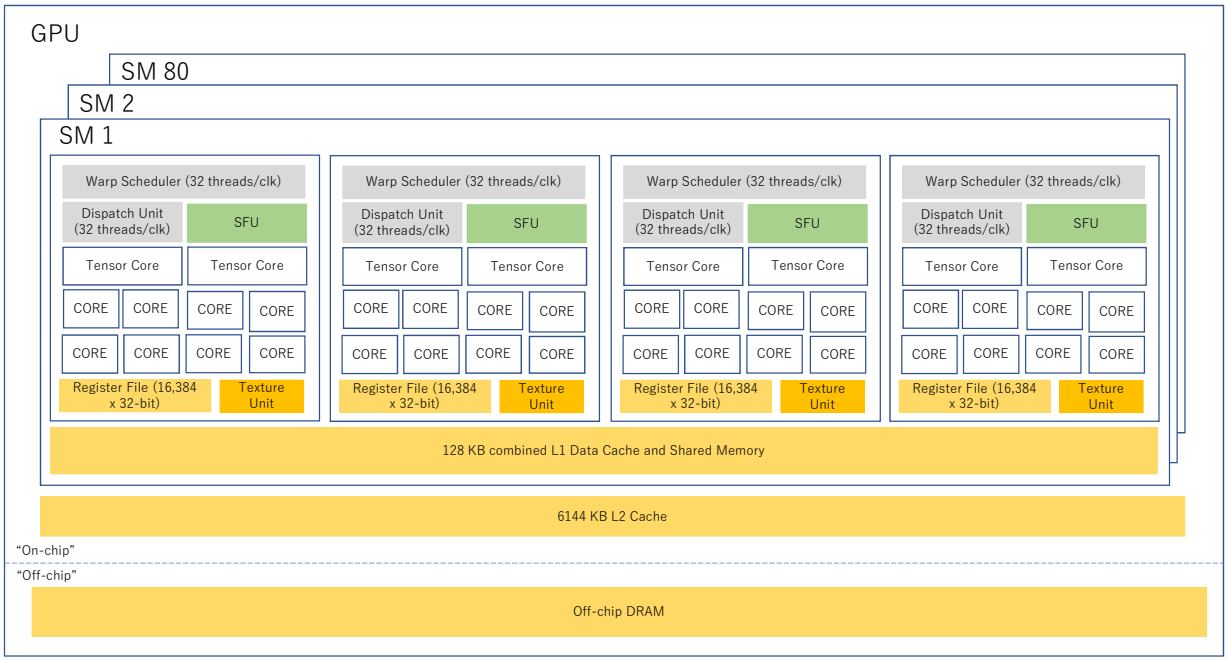

Figure 1.2: A simplified view of the architecture of an NVIDIA Tesla V100 GPU.

1.2 Overview of GPU

NVIDIA and AMD are the two leading vendors for commodity GPUs, meeting various demands from desktops to data centers. In this work, we focus on NVIDIA GPUs and their programming frameworks, namely, Compute Unified Device Architecture (CUDA) [65] and Open Accelerators (OpenACC) [70]. The reason is that NVIDIA GPUs yield the best performance per unit and thus lead both the accelerator market and the field of accelerator-based parallel computing. Nevertheless, the methods proposed in the thesis are vendor-agnostic.

1.2.1 Architectural Difference between CPU and GPU

The main reason for a CPU and a GPU to be different in architecture is that they are designed for different purposes. CPUs prefer tasks that are *narrow* and *deep*, while GPUs, on the other hand, prefer tasks that are *wide* and *shallow* [8]. In the architecture sense, a modern CPU is composed of a small number (less than 100) of powerful Arithmetic Logic Unit

(ALU) cores, lots of cache memory, one huge control module that can handle a few software threads at a time. The powerful CPU cores are equipped with complex circuits for functions like instruction reordering, branch prediction, out-of-order execution, paging, and caching. Therefore, CPUs can handle versatile tasks with low latency but limited concurrency. In contrast, a GPU has a great number (several thousand) of small compute cores with small control modules and small cache (available per core). Moreover, the GPU memory has an extremely high bandwidth compared to the CPU memory. GPUs thus excel in concurrently processing massive vectorizable data.

An NVIDIA GPU has a hierarchical architecture. Figure 1.2 describes the architecture of a Tesla V100 GPU [62], that is, compute (i.e., CUDA) cores are grouped into an array of streaming multiprocessors (SMs). In addition to the CUDA cores, each SM is equipped with special function units (SFUs), tensor cores, warp schedulers, dispatch units, and so forth. The CUDA cores are in charge of processing 32-bit integer, single- and double-precision floating-point operations. SFUs are in charge of special arithmetic operations like reciprocation. A tensor core, first proposed in NVIDIA's Volta architecture, is a novel unit to boost the process of training large artificial neural networks (ANNs) by enabling batched fused multiply-add (FMA) operations on small matrices [27]. Moreover, the NVIDIA GPUs pack every 32 threads into the smallest execution unit called *warp* and the warp schedulers and dispatch units schedule and dispatch warps to CUDA cores to execute.

1.2.2 GPU Computing Model

In terms of the computing model, the main difference between CPU and GPU is the granularity of parallelism. Processing the same amount of data, a GPU can exploit much more fine-grained data parallelism, a.k.a. single-instruction multiple-data (SIMD), than a CPU can because a GPU has hundreds of thousands of concurrent threads at runtime while a CPU has at most several hundred. Moreover, the constraint of the SIMD computing model adopted by CPUs is that CPUs use vector instructions (e.g., AVX for Intel CPUs) that require contiguously loading and storing data. NVIDIA proposed single instruction multiple threads (SIMT) model that relaxes the constraint of SIMD. The SIMT model allows threads in a warp to take different execution paths and thus enables thread-level parallel code for independent, scalar threads [65]. In this section, we use the V100 GPU (Figure 1.2) to explain the GPU computing model in detail.

First, GPU computing deals with a hierarchical memory system. Massive data parallelism is enabled by an array of SMs. Within each SM, there are registers and a combined L1 data cache and share memory. The 64K 32-bit register files provide the lowest access latency to compute cores. The combined L1 data cache and shared memory is a 128 KB memory

block that is configurable. For instance, if a program utilizes 32 KB shared memory, then 96 KB L1 cache is available for load/store operations. L1 cache is analogous to that of a CPU, preventing threads in an SM from accessing GPU memory. Shared memory is high-bandwidth and low-latency memory without cache misses, allowing threads in an SM to share data efficiently. To use shared memory requires writing code to explicitly manage the shared memory, which increases the programming effort. Outside of the SMs, there is L2 cache, the most outer on-chip memory, which works with the memory controller to handle memory requests to exploit data locality. In addition to the aforementioned on-chip memory, GPU has the off-chip DRAM, a.k.a GPU memory or GPU memory. Compared to the on-chip memory, the GPU memory has a relatively large capacity. For example, the V100 GPU has two configurations of the capacity of GPU memory, 16 GB or 32 GB. Despite the relatively large capacity, the GPU memory is much slower than the on-chip memory.

Secondly, GPU computing deals with a hierarchy of threads. When a parallel program runs on a GPU, thread blocks are cyclically assigned to SMs until all blocks finish execution. A *block* is a group of threads, which is a programmable execution unit to execute a compute kernel. Blocks must be mutually independent, that is, without any data dependency. Moreover, there is a max number of threads in a block, because an SM can only provide a finite amount of resources such as registers and shared memory. A block assigned to an SM is further scheduled and dispatched in warps as introduced in Section 1.2.1. Threads in a warp share the same instruction at each clock cycle and are processed in parallel in the SM. Moreover, to hide the high latency of access to GPU memory, a favorable approach is to assign multiple blocks to an SM, which overlaps access to GPU memory with computation for different warps.

Finally, although the SIMT model allows threads in a warp to store and load data in a non-contiguous manner, such an operation must be carefully used because it may violate the principle of memory coalescing and thus degrades the performance. Precisely, every successive 128 bytes memory, which is equal to the size of an L1 cache line, can be accessed by a warp in a single transaction on an NVIDIA GPU [61]. The length of the accessed memory space is called a stride. If the memory access requests from a warp have a stride larger than 128 bytes, the requests will be severed into multiple transactions. Other factors such as shared memory bank conflicts and overuse of diverging threads in a warp also impair the performance of parallel codes and should therefore be avoided in programming practices.

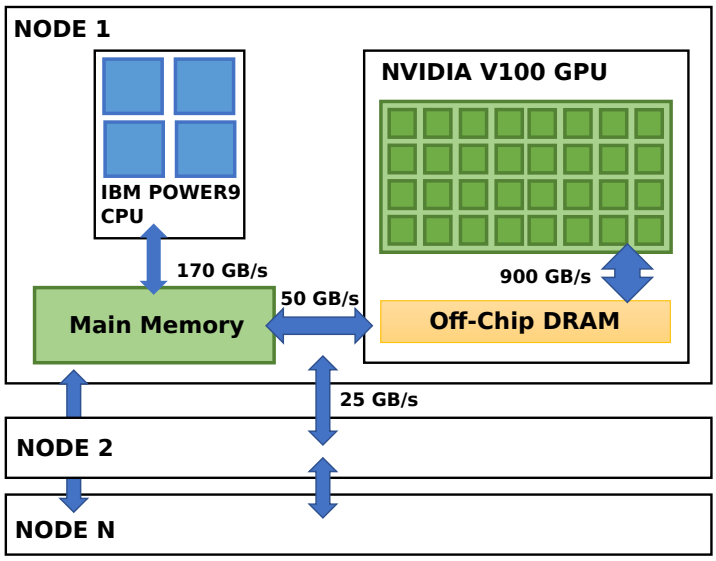

Figure 1.3: A simplified view of heterogeneous architecture, inspired by the architecture of Summit [40, 79]. Note that the bandwidth significantly varies in the hierarchy of memory.

1.3 Heterogeneous and Data-Centric Computing Techniques

Many state-of-the-art supercomputers such as Summit and Sierra [40] adopt a heterogeneous (i.e., CPU-GPU) architecture that has relatively few but *fat* nodes, compared to supercomputers with a homogeneous (i.e., CPU-only) architecture like Sunway TaihuLight [22]. A fat node has multiple CPUs and GPUs and thus outperforms a CPU-only node. Additionally, having few nodes is critical to the overall performance of a supercomputer because the inter-node data transfer is time-consuming. Figure 1.3 gives a simplified view of a supercomputer based on fat nodes. As you can see, such a heterogeneous architecture has various bandwidths between its components. Data transfers between nodes and between the CPU and GPU are extremely time-consuming compared to access from GPU compute cores to the GPU memory. According to a comprehensive survey, rapidly increased data volumes challenge the research and applications of exascale computing due to a growing gap between the computing power of processors and data bandwidth [28]. Therefore, a *data-centric* mindset is heavily relied on to create programs for heterogeneous architectures.

The data-centric computing concept merges innovative hardware and software to extract as much value as possible from existing and new data sources [100], emphasizing that computation should be performed where data resides especially for places where massively parallel computing is available [88, 109]. On this ground, we can consider a GPU as hardware

conforming to the data-centric computing concept, because a GPU utilizes high bandwidth memory and massive compute cores to apply massively parallel computing to the data residing on its memory. However, the capacity of GPU memory is relatively small (several tens of GBs). In the era of exascale computing, HPC data comes in petabytes (10^6 GB a.k.a PB) that dwarf the capacity of GPU memory. Such data cannot fit in the GPU memory and is thus called excess data.

To handle excess data, we need an *out-of-core* approach that decomposes the data into smaller chunks and then streams the chunks to and from the GPU to process. Such an approach relaxes the capacity limitation of GPU memory, but incurs more data transfer between the CPU and GPU, limiting the overall performance of the program.

In this work, we propose data-centric computing techniques for out-of-core GPU applications that mitigate the side effect involved by CPU-GPU data transfer. The techniques make as much use as possible of the on-GPU data, reduce as a large amount as possible of the CPU-GPU data transfer, and hide as much as possible the CPU-GPU data transfer time. Moreover, although we focus on reducing the data transfer between the CPU and GPU, the techniques are also applicable for reducing the inter-node data transfer.

1.4 Contributions of This Thesis

The contributions of this thesis are divided into three parts:

In the first part, we study a GPU-based solver for large instances of a combinatorial-optimization problem that generate excess data at runtime. We identify the CPU-GPU data transfer bottleneck in executing this out-of-core GPU application and thus propose three data-centric computing techniques that significantly mitigate the side effect involved by the data transfer.

In the second part, we investigate another important class of large-scale scientific applications, i.e., stencil computation. We find the data transfer bottleneck of the out-of-core GPU stencil codes generated by a previous framework [54]. Therefore, we propose two data-centric computing techniques to extend the framework that eliminates data copy on the CPU and reduces the CPU-GPU data transfer.

In the third part, as a complement to the second part, we propose one more data-centric technique that integrates on-the-fly GPU compression into the process of large GPU stencil computation, shrinking the data itself and thus reducing the amount of CPU-GPU data transfer. This technique contributes to a further performance improvement with a tolerable fidelity loss in the output.

Chapter 2

GPU-based Branch-and-Bound Method to Solve Large 0-1 Knapsack Problems with Data-centric Strategies

2.1 Introduction

The 0-1 knapsack problem [49], which is a combinatorial-optimization problem, occurs in a wide range of fields such as manufacturing, logistics, and bioinformatics [4, 71, 73]. Given n (≥ 1) items, each having profit and weight, the problem is to find a combination of items whose total weight does not exceed capacity c and total profit is maximized:

$$\text{maximize} \quad \sum_{i=1}^{n} p_i x_i, \quad \text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq c, \ i \in \{1, 2, \ldots, n\},$$

where p_i and w_i are the profit and weight of the i-th item, respectively, and $x_i \in \{0, 1\}$ is a binary decision variable that determines if the i-th item will be included (x_i=1 if included; otherwise, x_i=0). There are several approaches to the 0-1 knapsack problem. Among those, greedy [107] and genetic [30] algorithms are used to obtain feasible solutions whereas dynamic programming (DP) [3] and branch-and-bound (B&B) [43] approaches are for exact solutions. The DP approach was the first exact algorithm to solve 0-1 knapsack problem. The key idea of DP approach is to break down the original problem into a collection of simpler subproblems, solve each of those subproblems just once, and store the intermediate results using a table. The table size is subject to the capacity of knapsack and the number of items, and thus a high memory requirement is frequently cited as the main drawback of dynamic programming. On the other hand, the B&B approach is the most common way to effectively find the exact solutions for non-deterministic polynomial-time hard (NP-hard) combinatorial-optimization problems such as knapsack problems. An iterative B&B scheme (Figure 2.1) updates the upper and lower bounds at each iteration to reduce the search

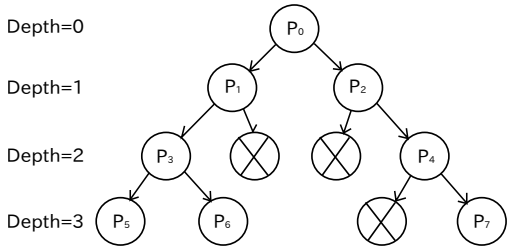

Figure 2.1: B&B search tree; root P_0 is the original problem, node P_i is the i-th subproblem, and X represents pruned subproblems (i.e., passive subproblems).

space. Each iteration comprises branching, bounding, and pruning operations. The branching operation decomposes a problem into multiple small subproblems. Then, the bounding operation computes the lower and upper bounds for each subproblem. Finally, the pruning operation discards unpromising (passive) subproblems whose upper bounds are less than the best solution found so far. The remaining promising (active) subproblems proceed to the next iteration and the process is repeated. B&B approaches can successfully solve knapsack problems by exploiting data parallelism using various parallel machines such as SIMD machines [44], cluster systems [17], computational grids [23, 93], and GPUs [7, 42, 69]. In addition to graphics applications, GPUs [47] are powerful accelerator devices for compute- and memory-intensive applications [33, 55, 106]. However, typically, GPU memory capacity, e.g., 16 GB, is relatively small compared to that of CPU memory. Consequently, previous GPU-based solvers [7, 42] failed to handle large problems that consume GPU memory rapidly when splitting a large problem into subproblems (e.g., a branching operation). Compared to these in-core methods [7, 42], we believe that out-of-core methods to solve the large 0-1 knapsack problems have not been studied sufficiently. In fact, excess data, (i.e., beyond memory capacity), is a major challenge for developing HPC applications. Many data-rich studies have adopted out-of-core approaches to relax memory capacity constraints [46, 53]. However, CPU-GPU data transfer overhead can become an implementation bottleneck, and thus, must be handled carefully. Furthermore, as stated in the literature [81], computation cost is improving at much higher rate than data movement cost; thus, we must shift from compute-centric to data-centric models. For example, to evolve the GPU-based implementations on the most recent architecture, CUDA [63] provides data-centric techniques, e.g., pipelining [91] to overlap CPU-GPU data transfer and GPU-based computation. However, as shown in Figure 2.2, GPU memory bandwidth improves "at much higher rate" [81] than PCIe bandwidth, broadening the gap between GPU memory bandwidth and PCIe bandwidth. Consequently, it becomes increasingly difficult to hide CPU-GPU data transfer overhead by overlapping data transfer with GPU computation. A possible solution is to use

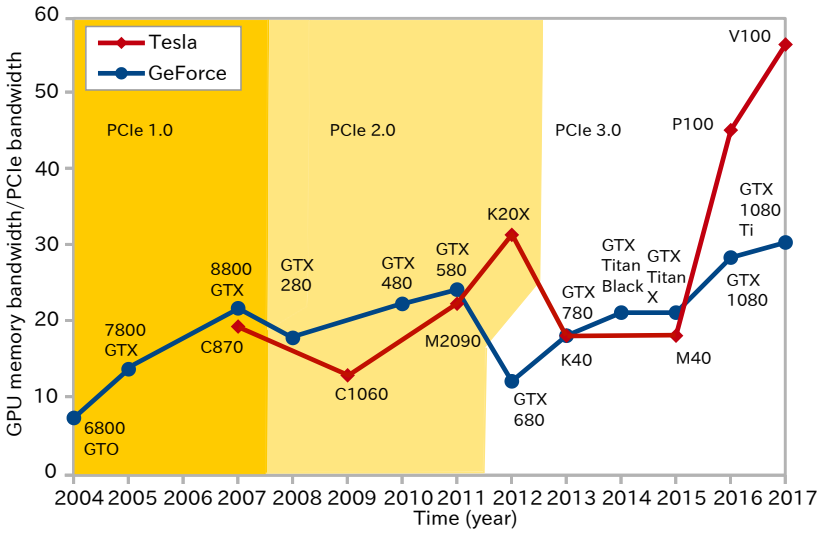

Figure 2.2: Gap between GPU memory bandwidth and PCIe bandwidth over time. For Tesla GPUs, the ratio of GPU memory bandwidth to PCIe bandwidth has increased by a factor of three within ten years.

the NVIDIA NVLink interconnect technology [62] rather than PCIe interconnect. NVLink provides interconnection between GPUs and between a CPU and GPU, and is considered 5–12 times as fast as PCIe relative to data transfer speed. However, it requires both specific NVLink-enabled CPU chipsets and GPUs, therefore, we consider that a software solution is preferable to a hardware solution.

In this paper, we propose an extension of our previous out-of-core method [84], such that it can hide data-transfer overhead when running on a latest GPU (Tesla V100 PCIe version). The proposed extension concentrates on using data-centric computing techniques (i.e., "strategies," used to distinguish Chapter 2 from Chapters 3 and 4), including introducing a data-centric search strategy, to evolve the previous method on the state-of-the-art machines. Compared to an aforementioned in-core method [42], the proposed out-of-core method relaxes the problem size limitation (i.e., maximum number of subproblems) from the capacity of GPU memory to that of CPU memory. To solve a large problem, the proposed method buffers subproblem data in CPU memory rather than GPU memory. However, such a CPU-centric subproblem management scheme increases CPU-GPU data transfer, which reduces GPU performance. Our main contribution over the in-core method [42] can be summarized as three data-centric strategies that eliminate the side effect incurred by CPU-GPU data transfer.

(1) O3S strategy. This strategy traverses the B&B search tree with less CPU-GPU data transfer. For each iteration of GPU B&B operations, an adaptive number of subproblems are transferred between the CPU and GPU according to GPU buffer usage. This

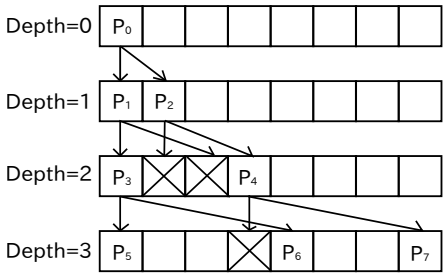

Figure 2.3: Array-based subproblem management scheme. Pruned subproblems (i.e., passive; X) make the array sparse as the depth of the search tree increases. Note that the branching, bounding, and pruning operations are performed on the GPU.

strategy can reduce the amount of data transfer significantly compared to our previous out-of-core method [84], which transfers all subproblems in a GPU buffer to apply a breadth-first search (BFS) strategy.

(2) Explicitly-managed pipelining strategy. Using this strategy hides data transfer overhead by overlapping data transfer with GPU-based B&B operations. We adopt this strategy instead of useful memory models such as unified and mapped memory [63] techniques. Although the latter techniques are useful for facilitating GPU programming, they cannot hide data transfer overhead in our case, which will be discussed in Section 2.5.2.

(3) GPU-based stream compaction strategy. This strategy converts sparse arrays into dense arrays to reduce the amount of data to transfer and achieve full utilization of massively parallel GPU cores. Here, we employ an input-output separated scheme that is more efficient than an input-output unified scheme [84]. In brief, the unified scheme introduces an overhead due to mapping the needed and unneeded elements in the input array to perform in-place movement.

The extended part of the proposed method over the base method [84] includes (1) the O3S strategy that helps hide data-transfer overhead when running on a latest GPU, (2) the explicitly-managed pipelining strategy designed in comparison with useful memory models, (3) the GPU-based stream compaction strategy improved further with the CUB library [67].

The remainder of this paper is organized as follows. Related work is discussed in Section 2.2. Parallelized GPU-based B&B methods are summarized in Section 2.3. The base [84] and proposed methods are described in Sections 2.4 and 2.5, respectively, and experimental results are given in Section 2.6. Conclusion and suggestions for future work are given in Section 2.7.

2.2 Related Work

Boukedjar *et al.* [7] presented a B&B approach that solves the 0-1 knapsack problem on a GPU. Their method manages subproblems in GPU memory with arrays representing subproblems, where a subproblem has several attributes, such as the upper and lower bounds of profit and the currently obtained profit and weight. They prepare an array for each attribute; however, those arrays may become sparse after B&B operations are performed (Figure 2.3) because elements representing passive subproblems are not considered in subsequent operations. This sparseness wastes GPU memory and negatively affects reference locality. Therefore, they integrate a CPU-based compaction strategy to reduce sparseness. The basic idea of their compaction strategy is to exchange passive and active elements using a quicksort-wise scheme such that all active subproblems are stored continuously at the front of the array, which is a CPU-based sequential operation; thus, their method transfers all attributes from GPU to CPU memory each time compaction proceeds. Therefore, CPU-GPU data transfer remains a performance bottleneck.

To reduce CPU-GPU data transfer, Lalami *et al.* [42] extended the aforementioned work [7] by restricting transferred data to a single attribute, i.e., label data (label array), where a label indicates whether an element is active ($label$=1) or passive ($label$=0). They focused on the data access pattern required for compaction (i.e., any attribute array generates the same pattern as those generated by other attribute arrays). As such, they compute the pattern on the CPU according to the label data and reuse that information to perform stream compaction for each attribute array on the GPU. This extension successfully improves the performance by a factor of two compared to the baseline method [7]. However, data are still transferred between the GPU and CPU each time stream compaction proceeds. Note that their method can solve large problems if subproblems are managed in CPU memory.

Shen *et al.* [84] presented a GPU-accelerated B&B approach to solve 0-1 knapsack problems that adopts a BFS strategy to traverse the B&B search tree. The BFS strategy treats all subproblems at the same depth of the search tree more fairly than other search strategies, e.g., a depth-first search strategy; however, the order of subproblems relative to their depths in the search tree must be maintained. Consequently, for each iteration, the BFS strategy must first transfer all the subproblems stored in a GPU buffer (i.e., intermediate results obtained in the previous iteration) to CPU memory, and then transfer a series of subproblems, whose amount is one-half the size of the GPU buffer, from CPU memory. Here the negative effect of increasing data transfer was not noticeable because the gap between GPU memory bandwidth and PCIe bandwidth was tolerable in their testbed (i.e., GTX Titan X and PCIe 3.0 $\times$16), which means that the GPU computation time was sufficiently long to completely

hide the CPU-GPU data transfer overhead. However, as explained in Section 2.1, the gap between GPU memory bandwidth and PCIe bandwidth has increased. For example, on a state-of-the-art GPU such as a PCIe Tesla V100, data transfer overhead becomes greater than GPU computation time. Consequently, performance is reduced significantly because the BFS strategy [84] cannot hide data transfer overhead. Therefore, Shen *et al.* [85] then extended [84] with three data-centric strategies to cope with such a data-transfer bottleneck.

Carneiro *et al.* [11] presented a GPU-accelerated B&B approach that uses a hybrid BFS and depth-first search (DFS) scheme to solve the symmetric traveling salesman problem. Their method initially performs BFS on the CPU to generate multiple initial subproblems to be examined in parallel. Then, their method switches to DFS to process B&B operations on the GPU. The behavior of our proposed O3S strategy is similar to their hybrid search strategy in that we use BFS to process subproblems if data do not need to be transferred between the CPU and GPU; otherwise, we ignore the BFS order, aiming at minimizing the CPU-GPU data (i.e., subproblems) transfer.

Silva *et al.* [89] presented a memory-aware load balancing strategy on a parallel B&B application. They attempted to solve the set partitioning problem on a cluster system. To obtain a feasible solution quickly, such that they can proceed to search the optimal solutions, they make each compute node process the DFS. In contrast, we propose an O3S strategy that produces a sufficient number of subproblems for the GPU to process and ensures sufficient GPU memory to store intermediate results temporarily.

Wan *et al.* [98] presented an efficient cooperative CPU-GPU computing method to solve the subset-sum problem. They implemented a parallel two-list algorithm that divides a set of n numbers into two smaller subsets and computes all possible subset-sums for these two subsets. During the subset-sum generating phase, they store all intermediate results that require $\mathcal{O}(2^{n/2})$ space in GPU memory. As a result, the problem size is limited by GPU memory capacity. Our proposed method relaxes this limitation because we use the CPU memory to buffer intermediate results (i.e., subproblems), which increase gradually; and results can be deleted if they are unpromising.

Quan *et al.* [72] presented a design and evaluation of a parallel neighbor algorithm for the disjunctively constrained knapsack problem. They proposed a parallel multi-neighborhood search method that simulates a teamwork that can find more feasible solutions gradually in the given timesteps. Note that our proposed method attempts to find optimal rather than feasible solutions.

Zavala-Díaz *et al.* [108] proposed a method that used multi-core CPU nodes to search partial B&B trees in parallel. Each partial B&B tree is part of the entire B&B search tree, i.e., a subproblem. BFS was used in their method because they use only CPUs for compu-

tation, suffering from no CPU-GPU data transfer. Nevertheless, GPU acceleration with our proposed O3S strategy can be integrated into their method to obtain further speedups.

Chen *et al.* [13] proposed BGPQ: an efficient GPU-based priority queue, exploiting both task and data parallelism. They found BGPQ outperformed the priority queues implemented by previous studies in solving 0-1 knapsack and A-star search problems. For solving a 0-1 knapsack problem, they used BGPQ to buffer subproblems on a GPU. Subproblems with higher values are sorted in the front of the queue and thus are searched earlier than subproblems with lower values. Nevertheless, using both CPU and GPU memory to manage subproblems was yet considered, and therefore our proposed out-of-core data management with O3S strategy can be beneficial for their method with respect to handling large problems that generate massive subproblems exceeding the capacity of GPU memory.

Ezugwu *et al.* [19] explored meta-heuristics in solving 0-1 knapsack problems. They found that both B&B and DP were the most effective approaches, whereas randomness-based approaches such as genetic algorithm and simulated annealing had major limitations: a trade-off between the quality of solution and search time and difficulty in ensuring to find the exact solution. In our work, we propose a GPU B&B approach to solve large 0-1 knapsack problems that involves data transfer between the CPU and GPU. An O3S strategy is thus proposed to search the subproblems while significantly reducing the CPU-GPU data transfer that limits the overall performance. Selection procedures that determine which subproblems are to be searched first are not considered in both baseline and proposed approaches because selection procedures are irrelevant to reducing the CPU-GPU data transfer per B&B iteration.

2.3 Parallelized B&B Approach to Solve Knapsack Problem

The B&B approach constructs a search tree that is rooted to the original problem in a way of dynamically splitting the promising nodes (i.e., subproblems) and deleting the unpromising nodes. Because the co-existing subproblems are mutually independent, we can process them in parallel. In this study, we implement the parallelized B&B approach with CUDA [63], an efficient parallel programming model for NVIDIA GPU chipsets.

2.3.1 B&B Approach Basics

In the B&B approach, the branching operation divides a single n-variable (i.e., item) problem into two (n-1)-variable subproblems. These two subproblems represent two cases for decision relative to the i-th item, where $1 \leq i \leq n$. For an n-item knapsack problem, from the first to

last items, we iteratively divide a single problem into two subproblems using an O3S strategy which is discussed in Section 2.5.1.

The bounding operation reduces the search space by determining whether it is possible to obtain an optimal solution for each subproblem. Here, the bounding operation computes the upper and lower bounds for all subproblems. A subproblem whose upper bound is less than the current lower bound is passive and must be deleted from the search space. In brief, the lower bound of a subproblem is obtained by picking items in a greedy manner, whereas the upper bound is obtained using the product of the reminder capacity and the greatest profit/weight ratio of remaining items. The current lower bound is the greatest lower bound found currently, which is updated per B&B iteration at runtime. Here, we assume that items are sorted according to decreasing profit per weight. This assumption facilitates the following lower bound computation. Let k be the index of the current item to be examined for decision and let I_v be the set of items picked for a subproblem, i.e., a vertex v in the search tree. The weight W_v and profit P_v of vertex v can then be computed as follows:

$$W_v = \sum_{i \in I_v} w_i, \tag{2.1}$$

$$P_v = \sum_{i \in I_v} p_i. \tag{2.2}$$

A vertex v can be described as a tuple $(W_v, P_v, U_v, L_v, S_v)$, where U_v and L_v represent the upper and lower bounds of the vertex, respectively, and S_v is the slack variable [42] for the vertex such that

$$\sum_{i=k+1}^{S_v-1} w_i \leq c - W_v < \sum_{i=k+1}^{S_v} w_i. \tag{2.3}$$

In other words, slack variable S_v is determined by picking all items after the k-th item as long as the knapsack can include them. Using the slack variable, the residual capacity r of the knapsack is given as follows:

$$r = c - W_v - \sum_{i=k+1}^{S_v-1} w_i. \tag{2.4}$$

Lower bound L_v can be obtained by picking the abovementioned items (i.e., from k+1 to S_v-1) and others in a greedy manner as follows:

$$L_v = P_v + \sum_{i=k+1}^{S_v-1} p_i + \sum_{i=S_v}^{n} p_i x_i, \tag{2.5}$$

where $x_i = 1$ if $w_i \leq r - \sum_{j=S_v+1}^{i-1} w_j x_j$. On the other hand, relative to an upper bound U_v, we adopt the Dantzig bound [14], which is expressed as follows:

$$U_v = P_v + \sum_{i=k+1}^{S_v-1} p_i + \lfloor r P_{S_v} / W_{S_v} \rfloor. \tag{2.6}$$

2.3.2 Parallel Programming with CUDA

NVIDIA's CUDA [63] is the most prevalent parallel computing platform and application programming interface model for developing GPU-accelerated applications. The CUDA parallel programming model is designed for programmers familiar with standard programming languages such as C. At the core of model there are three key abstractions - a hierarchy of thread groups, shared memories, and barrier synchronization which are exposed to the programmer to define how to parallelize his job. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse subproblems that can be solved independently in parallel by blocks of threads, and each subproblem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

In more detail, CUDA C extends standard C by allowing the programmers to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. In our case, we defined three kernels to perform B&B operations: branch kernel divides each subproblem into two smaller subproblems; bound kernel computes the upper and lower bound for each subproblems; and compact kernel prunes unpromising subproblems by overwriting them with promising subproblems.

2.4 Base Method

Our base method [84] employs a CPU-centric subproblem management scheme that exploits the large capacity of CPU memory to relax the problem size limitation. Moreover, we integrated a GPU-based stream compaction strategy into the method to reduce the sparsity of data before transferring it between the CPU and GPU.

2.4.1 CPU-Centric Subproblem Management

To realize efficient CPU-centric subproblem management, the data structure that stores subproblems must satisfy two requirements.

(1) The data structure must be accessed randomly by a GPU thread in $\mathcal{O}(1)$ time.

(2) The data structure must store subproblems continuously without fragmentation to improve the efficiency of direct memory access (DMA) required for CPU-GPU data transfer.

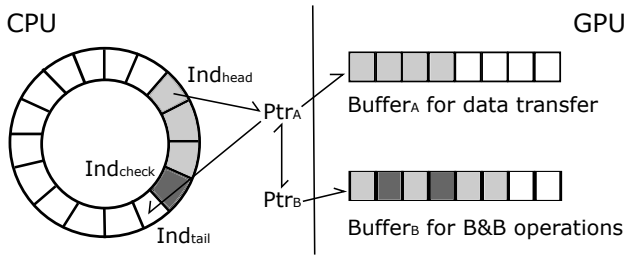

Figure 2.4: Overview of CPU-centric subproblem management. Double buffering is used to overlap data transfer with GPU computation.

To satisfy requirement (1), we select arrays as a buffer that stores subproblems for both CPU memory and GPU memory. To satisfy requirement (2), we organize the CPU buffer in a circular manner with three indices that instruct to read and write data (i.e., subproblems) continuously.

As shown in Figure 2.4, Ind_{head} denotes the index of the first subproblem in the buffer (i.e., array), Ind_{tail} denotes the index behind the index of the last subproblem in the buffer, and Ind_{check} denotes the index of the first finished subproblem in the buffer. During execution, the proposed method transfers a series of subproblems that begin with the subproblems referenced by Ind_{head} from CPU memory to GPU memory in order to process the GPU-based B&B operations. On the other hand, when predicting whether GPU memory will be exhausted after the next branching operation (i.e., doubling the number of subproblems), the proposed method transfers the subproblems from GPU memory to CPU memory and stores them from the position referenced by Ind_{tail}. Each time a data transfer operation proceeds, the proposed method updates Ind_{head} or Ind_{tail} (according to the operation type) by an increment equal to the amount of transferred subproblems. The proposed method uses Ind_{check} to determine if the termination condition is satisfied, which is explained later in this section. Moreover, the circular manner means that if we reach the last position in the array while transferring data, we return to the first position in the array. However, in this case, we must call the data transfer function again because the baseline address has changed.

These transfer and computation jobs proceed iteratively until all subproblems are finished, meaning that a decision of the last item, (i.e., x_n), is performed for each subproblem. Furthermore, for each iteration, if some of the subproblems transferred from the GPU to CPU has not finished, the proposed method makes Ind_{check} equal to Ind_{tail}. Otherwise, Ind_{check} keeps its current value. If Ind_{check} is equal to Ind_{head}, the proposed method determines that the termination condition has been satisfied, (i.e., the B&B search tree has been searched completely). The proposed method then outputs the information (weight, profit, etc.) of all subproblems left in the CPU buffer as solutions to the knapsack problem

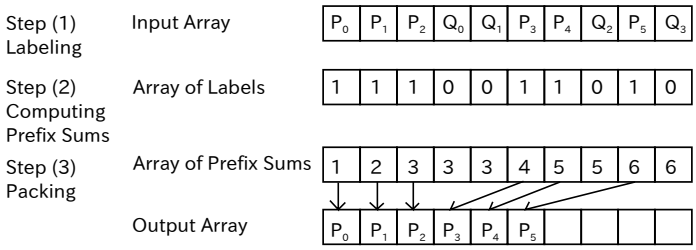

Figure 2.5: Overview of the separated stream compaction scheme. P_i refers to an active subproblem, and Q_i refers to a passive subproblem. In step (1), we associate active and passive subproblems with 1 and 0, respectively. In step (2), we apply the computation of prefix sums to the array of labels to obtain the array of prefix sums that indicates where to move the active subproblems (i.e., the data access pattern). In step (3), we contiguously pack active subproblems into the output array according to the array of prefix sums.

(Figure 2.4).

In addition, the proposed method chooses between two execution modes dynamically according to the total number of subproblems managed by both the CPU and GPU. If the circular buffer lacks a sufficient number of subproblems to fully exploit the massive parallelism of the GPU, we only use the CPU to process the branching, bounding, and pruning operations (otherwise, the GPU is used). In addition to this parallelism consideration, changing to CPU mode also helps determine whether the termination condition has been satisfied by processing subproblems sequentially. Relative to the timing of changing execution modes, we experimentally determined $24,576$ as the threshold number (i.e., the minimum number of subproblems required by the GPU execution mode) by comparing the sequential execution time to the CPU-GPU communication time.

2.4.2 GPU-Based Stream Compaction Strategy

To avoid extra CPU-GPU data (i.e., label array) transfer when computing the data access pattern required for compaction, we adopt a stream compaction strategy that proceeds entirely on the GPU. This stream compaction strategy computes the destinations required to move subproblems on the GPU rather than the CPU. As discussed in Section 2.1, we compared two stream compaction variations, a separated scheme and a unified scheme, and we found that the separated scheme is more efficient relative to execution time. Details about the unified scheme can be found in the literature [84]. The separated stream compaction is a filtering process and requires the input array (i.e., the array to be compacted) is separated from the output array. The elements that meet the requirements are selected from the input array and then packed into the output array. Generally, the compaction process is performed in three steps (Figure 2.5): (1) preparation of the array of labels, (2) computation of prefix sums, and (3) the packing procedure.

Table 2.1: Comparison of O3S and BFS [84] strategies in the most data transfer demanding case (i.e., no subproblems are pruned). For simplicity, we assume the GPU buffer begins with four subproblems and can store eight subproblems, and that the CPU buffer has sufficient subproblems to refill the GPU buffer. Note that DtoH and HtoD denote GPU-to-CPU (device-to-host) and CPU-to-GPU, respectively.

	BFS			O3S		
Operation	Buffer usage (sub-problems)	Transfer amount (sub-problems)	Transfer function call (times)	Buffer usage (sub-problems)	Transfer amount (sub-problems)	Transfer function call (times)
B&B	8	0	0	8	0	0
Pruning	8	0	0	8	0	0
DtoH transfer	0	8	1	4	4	1
HtoD transfer	4	4	1	4	0	0

2.5 Proposed Data-Centric Computing Strategies

We extend the aforementioned base method on the latest GPUs to deal with the broadening gap between the GPU memory bandwidth and PCIe bandwidth, which is a performance bottleneck of a GPU application. We eliminate the bottleneck by proposing three data-centric strategies: (1) an out-of-order search (O3S) strategy which reduces the amount of data transfer per iteration compared to the previous BFS strategy [84], (2) an explicitly-managed pipelining strategy which exposes the details of overlapping data transfer and GPU computation, and (3) a GPU-based stream compaction strategy improved further with the CUB [67] library. We also give some preliminary experimental results to conclude that single-threaded CPU computation is sufficient for maximizing the performance of our GPU-accelerated solver.

2.5.1 O3S Strategy

As explained in Section 2.2, the previous BFS strategy must maintain the order of subproblems relative to their depth in the search tree, which increases CPU-GPU data transfer sharply. As a result, the BFS strategy becomes ineffective because the cost of data transfer via PCIe buses becomes increasingly intolerable over time. In fact, during execution, all subproblems that exist simultaneously are independent; even if they are at different depths in the search tree. Therefore, there is no inherent requirement to maintain the order of the subproblems. Given this, we propose an O3S strategy to traverse the search tree that processes the subproblems regardless of order.

Algorithm 1 describes the behavior of the proposed O3S strategy. For each iteration of GPU B&B operations, the O3S strategy transfers an adaptive number of subproblems

between the CPU and GPU according to GPU buffer usage. In other words, if the number of subproblems in the GPU buffer is greater than one-half the size of the GPU buffer, the proposed strategy transfers an amount of subproblems back to CPU memory. If the number of subproblems in the GPU buffer is less than one-half the size of the GPU buffer, the proposed strategy transfers an amount of subproblems from the CPU memory. Otherwise, the proposed strategy continues to process the subproblem currently stored in the GPU buffer without transferring data. By adopting this O3S strategy, we reduce the amount of transferred data by two-thirds and data transfer function calls by one-half, even for the worst case, which the greatest number of data transfers (Table 2.1). Note that both O3S and BFS strategies transfer subproblems that are stored contiguously in the CPU and GPU buffers, instead of selecting specific subproblems to transfer. After pruning operation, the GPU buffers will be compacted to avoid transferring any unpromising subproblem.

In addition to reducing the amount of data transfer per iteration, the proposed O3S strategy improves robustness against a rapidly increasing number of subproblems because the behavior of this strategy lies between that of BFS strategy and DFS strategies, i.e., the proposed O3S strategy tends to finish searching some sub-trees of the entire search tree, which, to some extent, suppresses a quickly increasing number of subproblems.

However, a potential downside is that the proposed O3S strategy is less fair than the BFS strategy, i.e., the BFS strategy finds the lower bound for each entire depth it has traversed before it proceeds to the next depth. In contrast, the proposed O3S strategy may dig deeper while delaying processing some highly promising subproblems that have good lower bounds, which impairs the efficiency of pruning unpromising subproblems.

2.5.2 Explicitly-Managed Pipelining Strategy

CUDA provides several solutions to deal with large data that exceeds the capacity of GPU memory: (1) explicitly-managed memory, (2) mapped memory, and (3) unified memory. Among these solutions, we decided to deploy the first solution (i.e., explicitly-managed memory) to hide the CPU-GPU data transfer overhead. That is, we realize a double buffering strategy that overlaps the CPU-GPU data transfer with GPU-based B&B operations using two CUDA streams. Here, a CUDA stream is a sequence of operations that execute in issue-order on the GPU [91]. The data transfer proceeds on a stream using $buffer_A$, and the GPU-based B&B operations proceed on another stream using $buffer_B$. When operations on both streams are complete, the strategy exchanges the references (i.e., Ptr_A and Ptr_B in Figure 2.4) to $buffer_A$ and $buffer_B$, and proceeds to the next iteration of this overlapped transfer and computation procedure until finishing processing all subproblems remaining in the circular buffer. However, prior to the next iteration, the strategy must synchronize both

Algorithm 1: O3S strategy

Input: (1) s_{cpu}, CPU buffer size, (2) s_{gpu}, GPU buffer size, (3) *threshold*, minimum number of subproblems required for GPU execution mode

Output: buf_{cpu}, CPU buffer that manages subproblems and thus contains the results finally

```
1  Allocate memory for buf_cpu and buf_gpu (GPU buffer that contains intermediate
   results)
2  Add the original problem to buf_cpu
3  n_cpu ⇐ 1; n_gpu ⇐ 0
4  ind_head ⇐ 0; ind_tail ⇐ 1; ind_check ⇐ 1
5  while ind_head ≠ ind_check do
6      if n_cpu + n_gpu < threshold then
7          Transfer n_gpu subproblems from buf_gpu to buf_cpu
8          Update ind_end, ind_check, n_cpu, and n_gpu
9          Process the subproblems in buf_cpu on the CPU
10         Update ind_end, ind_check, and n_cpu
11     else if n_cpu + n_gpu ≥ threshold then
12         if n_gpu > s_gpu/2 then
13             if All the subproblems in the GPU buffer finished then
14                 amount ⇐ n_gpu
15                 if amount + n_cpu > s_cpu then
16                     Exit due to the CPU buffer being exhausted
17                 else
18                     Transfer amount subproblems from buf_gpu to buf_cpu
19                     Update ind_end, ind_check, n_cpu, and n_gpu
20             else
21                 amount ⇐ n_gpu − s_gpu/2
22                 if amount + n_cpu > s_cpu then
23                     Exit due to the CPU buffer being exhausted
24                 else
25                     Transfer amount subproblems from buf_gpu to buf_cpu
26                     Update ind_end, n_cpu, and n_gpu
27         else if n_gpu < s_gpu/2 then
28             amount ⇐ min(s_gpu − n_gpu, n_cpu)
29             Transfer amount subproblems from buf_cpu to buf_gpu
30             Update ind_head, n_cpu, and n_gpu
31         if n_gpu ≠ 0 then
32             Process the subproblems in buf_gpu on the GPU Update n_gpu
```

streams to ensure that the current iteration has finished. Note that we perform synchronization to prevent both streams from transferring data simultaneously; otherwise, the circular buffer may become inconsistent. Moreover, to handle large data, we use cudaMallocHost() and cudaMalloc() to allocate pinned memory for the CPU and GPU buffers, respectively. We adopt the aforementioned double buffering to overlap CPU-GPU data transfer with GPU computation. In this strategy, we can determine the timing for transferring data between the CPU and GPU because we explicitly judge if the number of subproblems will exceed the size of GPU buffer; thus, we can overlap data transfer with GPU computation.

As for the second solution (i.e., mapped memory), this capability allows CUDA programmers to share page-locked CPU memory between CPU and GPU. With this capability, there is no need to allocate any GPU memory and explicitly copy data between GPU and CPU memory [63]. Accordingly, users can easily process large data whose size is beyond the capacity of GPU memory. However, owing to the lack of data copy function calls, data transfer cannot be explicitly overlapped with kernel execution.

Similar to mapped memory, unified memory facilitates programming tasks for dealing with large data on the GPU. Unified memory provides a common address space that can be accessed from hosts and devices with a coherent manner. Recent GPU architectures such as Pascal and Volta can allocate more memory than the physical size of GPU memory. However, the physical location of data is invisible to a program, which avoids explicit overlapping of data transfer and kernel execution. To deal with this data locality issue, CUDA provides useful APIs such as cudaMemPrefetchAsync() which allow users to prefetch pageable memory to the GPU. However, if we use a buffer allocated with unified memory, we cannot determine which part of the buffer is on the GPU and which part is on the CPU. Thus, we cannot know when to call the data transfer API.

Finally, dynamic parallelism [63] is also useful for reducing the amount of data transfer between CPU and GPU. This capability allows GPU threads (instead of CPU threads) to launch a kernel at runtime, and thus, reducing the need to transfer execution control and data between CPU and GPU. Dynamic parallelism is useful to implement a parallel recursive algorithm. Our algorithm can also be implemented with dynamic parallelism, but we avoid using this implementation strategy because recursive algorithms easily exhaust computational resources such as register files and shared memory. In a recursive algorithm, parent threads will not complete before child threads, so that resources are rapidly wasted as the recursion goes deeper. To make the matters worse, we have to reduce the sparseness of array data to maximize GPU acceleration. Therefore, we adopt a synchronous approach that launch a kernel from the CPU.

2.5.3 CUB-Based Stream Compaction

We compare the Thrust [35] with CUB [67] library to find a better way to implement the GPU-based stream compaction strategy. Both the Thrust and CUB libraries provide GPU-accelerated max reduce function and inclusive scan function. Max reduce function is used to find the optimum in the GPU buffer, and inclusive scan function is used to compute prefix sums of the label array. Moreover, both libraries provide similar device-wide primitives for CUDA. However, they target different abstraction layers for parallel computing. The Thrust abstractions are agnostic of any particular parallel framework (CUDA, TBB, OpenMP, sequential CPU, etc). While the Thrust library has a backend for CUDA devices, its interface is more abstract than that of the CUDA-specific CUB library. As such, Thrust interface avoids explicitly exposing CUDA-specific details (e.g., cudaStream_t parameters). On the other hand, the CUB library is specific to CUDA C++ and its interface accommodates CUDA-specific features explicitly. For example, CUB uses shuffle instructions with predicate to realize prefix sum computation, whereas Thrust uses shuffle instructions without predicate. Moreover, CUB exposes more details of execution to its users; thus, the users have more freedom to tune their programs implemented with CUB. An example is to perform prefix sum computation. CUB allows its users to designate a temporary storage in the GPU memory for holding the intermediate results (i.e., partial sums). In contrast, Thrust hides this detail by allocating the temporary storage for the users. The cost of this memory allocation operation is tolerable, if a user calls the function only once. However, in our case, we have several thousand iterations when solving a large knapsack problem and each iteration requires to perform prefix sum computation. Therefore, we can allocate the temporary storage at the beginning of execution, and then pass its pointer to the prefix sum computation function provided by CUB. By doing so, we can avoid allocating the temporary memory for several thousand times. However, along with more freedom to control the execution, users also have more responsibility to the proper behaviors of program. In the prefix sum example, if the user designates a temporary storage whose size is insufficient, the function provided by CUB will return incorrect results. In our case, we can determine a sufficient size by referring to the size of the GPU buffer and that of the CUDA block.

2.5.4 CPU Threading Design

As we described in Section 2.4.1, we set the threshold for switching CPU/GPU execution modes by comparing the sequential execution time to the CPU-GPU communication time. However, because the CPU has multiple cores, we may be able to improve the performance of program by adopting CPU multi-threading techniques. Therefore, we have performed a

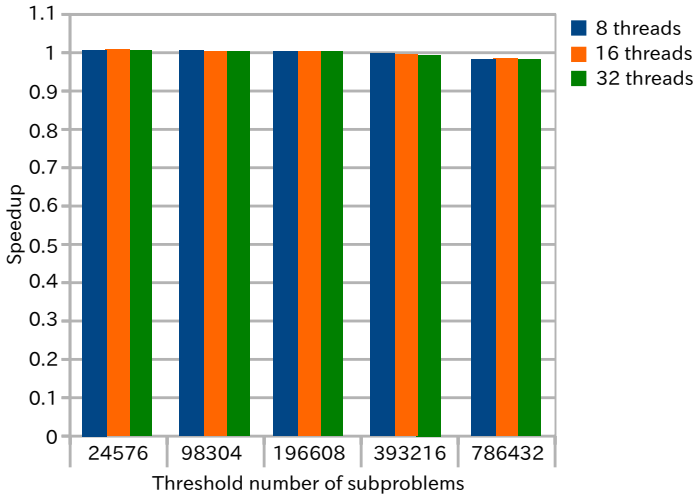

Figure 2.6: Speedup of multi-threading CPU mode over single-threading CPU mode. No significant speedup was achieved by multi-threading CPU mode.

preliminary experiment to determine whether we should implement CPU multi-threading techniques in our method.

In detail, we use OpenMP [96] 4.0 to implement CPU multi-threading. We use the single-threading CPU mode program as the baseline. As for the multi-threading version, we change the values of two variables, i.e., the number of threads and threshold number of subproblems, to generate multiple variations. We compare the single-threading with multi-threading programs by solving problems of items ranged from 600 to 1000. Figure 2.6 shows the results of solving the problem of 700 items. The results demonstrate that CPU multi-threading is unnecessary in our case. Neither the speedup nor degradation is significant if we apply CPU multi-threading to CPU execution mode with different threshold numbers. The main reason for the similar performance of the CPU single-threading and multi-threading programs is clear. The CPU mode execution time is a small proportion (less than 5% on average) of the total execution time when solving a large problem that incurs CPU-GPU data transfer for many times. Even we process hundreds of thousands subproblems in parallel on the CPU, the amount of work still pales compared to tens of millions subproblems processed in parallel on the GPU. Nevertheless, there is a lesser reason. A sufficient number of subproblems is needed for CPU multi-threading; otherwise, the cost of generating and controlling multiple threads overweighs the advantage of parallel execution. Even we set a sufficient number as a CPU mode threshold, we cannot ensure the number of subproblems to be near this threshold number, because the CPU execution mode is triggered whenever the number of subproblems is smaller than the threshold number. In contrast, GPU execution mode is triggered only if the number of subproblems is greater than the threshold number,

Table 2.2: Comparison of experimental methods to be evaluated.

	In-core [42]	Previous [84]	Extended
Data management scheme	In-core	Out-of-core	Out-of-core
Search strategy	BFS	BFS	O3S
Explicitly-managed pipelining	×	◯	◯
Stream compaction strategy	CPU-based	GPU-based, Thrust [35]	GPU-based, CUB [67]

Table 2.3: Experimental datasets.

Parameter	Value
p_i: profit of the i-th item	$w_i + 1000 + \text{random}(-20, 20)$
w_i: weight of the i-th item	$\text{random}(1, 10000)$
c: knapsack capacity	$\sum_{i=1}^{n} w_i * 100/1001$

which ensures that GPU always has sufficient number of subproblems to process.

2.6 Experimental Results

We conducted an experiment to evaluate the extended method in terms of problem size and execution time. Here, we used Lalami's in-core method [42] and our previous method [84] as baseline methods. Table 2.2 shows what techniques the extended and baseline methods comprise. All methods were compiled with the same optimization options. Moreover, for the large problems that the in-core method [42] failed to solve, we compared the extended method to our previous method [84] in terms of execution time and efficiency of finding the optimum to evaluate the performance of the O3S strategy. Furthermore, we analyzed the suitability of the extended method for machines with different ratios of GPU memory bandwidth to PCIe bandwidth. Finally, we compared the Thrust and CUB libraries in terms of execution time, aiming at improving the stream compaction strategy.

We used a suite of benchmarking datasets [50, 51] shown in Table 2.3. Here, we used strongly correlated instances with up to 1000 items, where the weight of each item primarily determines its profit. Such instances can be parallelized efficiently because of the enormous combinations of items that may be an optimal solution; therefore, such instances represent the most difficult problems. Note that we generated 20 unique instances for each value of n.

Table 2.4 shows the specifications of the experimental machines, which ran Ubuntu 16.04 with CUDA 9.0 [63]. The version of the Thrust library was 1.8.3. and that of the CUB library was 1.7.0. Moreover, for all experiments, we allocated an array of 20 GB for the CPU circular buffer, and we allocated 2 GB of GPU memory for the GPU buffer. In other words, for the in-core method, we allocated a single 2 GB buffer, and, for the previous and

Table 2.4: Testbed specifications. Ratio of GPU memory bandwidth to PCIe bandwidth of Machine 1 is 56 and that of Machine 2 is 21.

	Machine 1	Machine 2
CPU	Intel Xeon Silver 4110	Intel Xeon E5-2660 v3
CPU memory capacity	512 GB	64 GB
GPU	Tesla V100 PCIe (Volta)	GeForce GTX Titan X (Maxwell)
GPU memory capacity	16 GB	12 GB
GPU driver version	390.30	375.39
PCIe	gen3 ×16	gen3 ×16

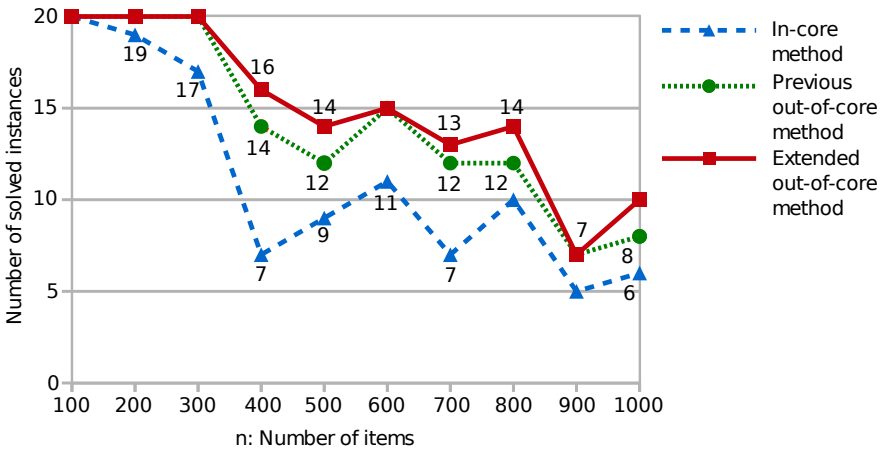

Figure 2.7: Comparison of numbers of instances solved by the extended, previous [84], and in-core [42] methods.

extended methods, we allocated two 1 GB buffers (double buffering).

2.6.1 Robustness against Increased Problem Size

We prepared 10 classes of instances, varying from 100 to 1000 items, and we generated 20 different instances for each class. Figure 2.7 shows that the extended method solved the greatest number of instances, thereby demonstrating that the O3S strategy is more robust than the BFS strategy [84]. This robustness occurred because the O3S strategy tended to finish searching some sub-trees before finishing the entire search tree, which reduced the tendency of splitting subproblems to some extent. For example, in the instance class with 400 items, the extended method solved two more instances than the previous method, denoted by $Inst_{13}$ and $Inst_{15}$. Here, we analyze $Inst_{13}$ because it produced more subproblems than $Inst_{15}$. The previous method terminated after the total number of subproblems reached $815,970,178$. On the other hand, at the same number of iterations, the extended method reached only $99,413,303$ subproblems in total, which suggests that the extended method reduced the tendency of splitting subproblems by a factor of eight. Furthermore, the extended method reached at most $204,725,088$ throughout execution when solving $Inst_{13}$, which is one-fourth the number of subproblems reached when the previous method terminated. Note that the extended method suppresses subproblem splitting only by changing the search order (i.e., adopting the O3S strategy), and in no case does the extended method omit a subproblem that should be searched. According to the results, robustness achieved by the extended method is noticeable, especially for properly large instances (i.e., 400 to 800 items). However, there was no significant difference for instances with greater than 800 items because the rapidly increasing number of subproblems exhausted CPU memory capacity. Although the number of solvable instances of proposed O3S-based method increased compared to that of previous BFS-based method, such an increase is not significant because the O3S strategy does not sharply improve the efficiency in avoiding CPU memory exhaustion compared to the BFS strategy. Note that the main contribution of the proposed O3S strategy is reducing the execution time by significantly reducing the amount of CPU-GPU data transfer.

In summary, the extended method buffered $41\times$ as many subproblems as the in-core method. For the solved instances, the extended method successfully stored $689,337,946$ subproblems at a time during execution, whereas the in-core method at most stored $16,783,230$ subproblems at a time. As a result, the extended method solved approximately twice as many instances as the in-core method.

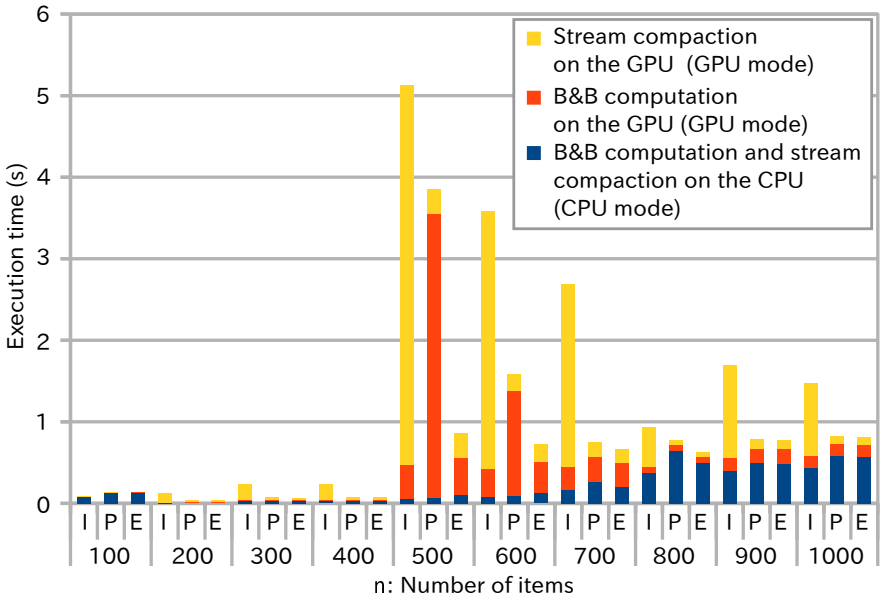

Figure 2.8: Comparison and breakdown of execution time for small instances. "I," "P," and "E" refer to the in-core [42], previous [84], and extended methods, respectively. The proposed stream compaction strategy outperformed the previous stream compaction strategy [42].

2.6.2 Evaluation of GPU-Based Stream Compaction Strategy

We compared the proposed stream compaction strategy to that of the in-core method [42] in terms of execution time (Figure 2.8). For each instance class, we selected a representative instance whose execution time was closest to the mean value of the execution time of all solved instances. Note that we ignored instances whose maximum number of subproblems stored at a time during execution was less than the threshold required for the GPU execution mode. Such instances avoided evoking GPU kernels; thus, they were useless for measuring the performance of the GPU-based B&B and stream compaction operations. Moreover, in this experiment, we used the CUB library to implement the prefix sum computation for the stream compaction strategy.

According to the results, both the extended and previous [84] methods achieved higher performance because the proposed stream compaction strategy was processed completely in parallel on the GPU . In contrast, as discussed in Section 2.2, the previous stream compaction strategy computed data access pattern sequentially on the CPU, and it must transfer the label array between the CPU and GPU. Therefore, the proposed GPU-based stream compaction strategy ran 9.9× faster on average than the previous strategy, obtaining an average speedup of 3.1× relative to total execution time. Note that because both in-core and out-of-core methods use CPU and GPU, the power consumption is determined by execution time. On this ground, the proposed out-of-core method is better than the in-core method in terms

of power efficiency.

Moreover, those instances can be solved by the previous in-core method; therefore, many of them (e.g., n=400, 900, and 1000) produced only a few subproblems, which avoided swapping subproblems between the CPU and GPU. For such small instances, the previous and extended methods preformed equally. On the other hand, for n=500, which produced a sufficient number of subproblems to trigger CPU-GPU data transfer, the extended method was 4.5× as fast as the previous method because the former method required only 13% of the GPU computation time of the latter method. We attribute this speedup to the O3S strategy because it significantly reduced the CPU-GPU data transfer overhead per iteration, which we discuss in Section 2.6.3.

Execution time appeared to be subject to a normal distribution against the number of items rather than a linear correlation. It is surprising that execution time required to solve a mid-sized problem (e.g., n=500) was greater than that for a large problem (e.g., n=900). However, this can be explained by survivorship bias. Regardless of pruning, the 900-item instances inherently produced more subproblems (2^{900}) than the 500-item instances (2^{500}). As such, the 900-item instances were prone to reaching a huge number of subproblems at a much earlier stage than the 500-item instances, due to the exponential correlation of item and subproblem numbers. As a result, many time-consuming 500-item instances survived (i.e., solved); however, no time-consuming 900-item instances survived. In other words, the survived 900-item instances are not large instances in fact. Because they failed to keep producing many subproblems exceeding the threshold number for GPU mode throughout the execution. Note that there is evidence for this deduction. For 900-item instances, CPU mode execution time represented a large proportion (62% for the extended method) of total execution time. Efficient pruning suppressed a large number of subproblems that exceed the GPU mode threshold, thereby proving that such instances were not time-consuming.

Moreover,

2.6.3 Evaluation of O3S Strategy

For large instances that the in-core method [42] failed to solve, we compared the extended method to the previous method [84] in terms of execution time and efficiency of finding the optimum. We selected a representative instance for each class as described in Section 2.6.2.

Figure 2.9 shows a breakdown of the execution time of the representative instances. According to the results, compared to the previous method, the extended method achieved a speedup of 11.7× (n=400) at most and 7.5× on average. This speedup was exclusively attributed to the O3S strategy because both the extended and previous methods implemented the same stream compaction strategy; therefore, no improvement came from that

part. The O3S strategy reduced the CPU-GPU data transfer per iteration significantly such that the data transfer overhead could be hidden by the overlapping GPU computation time. In contrast, the BFS strategy suffered from an unhideable data transfer overhead due to a huge amount of data transfer per iteration. Consequently, the GPU computation (which was overlapped with the CPU-GPU data transfer) of the extended method achieved a speedup of 23.5× at most (12.2× on average) compared to that of the previous method. For example, when solving the instance with 400 items, the GPU B&B computation time of the extended method was only 5% of that of the previous method.

Figure 2.10 shows how many B&B iterations were required for the BFS strategy (i.e., the previous method) and the O3S strategy (i.e., the extended method) to find the optimum and finish searching the B&B tree. As can be seen, the BFS strategy outperformed the O3S strategy because the BFS strategy can find the solution with a smaller number of iterations than the O3S strategy. Compared to the O3S strategy, the BFS strategy reduced the number of iterations by 4% on average and 7% at most (n=500) required to search the B&B tree completely, because the BFS strategy could find the optimum at an earlier stage than the O3S strategy. For example, when solving the instance with 400 items, the BFS strategy required one-eighth the number of iterations to find the optimum compared to the O3S strategy. However, because we used strongly correlated (i.e., hardest) instances, it was impossible to make an assertion at an early stage even if the optimum had already been found. Therefore, this merit of the BFS strategy barely made a significant contribution (i.e., a 7% reduction of total iterations at most), which was completely negligible compared to the reduction of data transfer per iteration achieved by the O3S strategy. Moreover, to compare the amount of work more precisely, we analyze the number of subproblems that have been processed when the programs terminated (Figure 2.10). According to the results, the difference of workload between the two strategies is trivial. As such, the difference of workload has little influence on the difference of performance between the two strategies. For example, the instances where the difference of workload is most observable are $n = 400$ and $n = 600$. However, in $n = 400$, the BFS strategy processed 52.81 billion subproblems while the O3S strategy processed 53.96 billion subproblems, where the difference is 2.1%; in $n = 600$, the BFS strategy processed 25.80 billion subproblems while the O3S strategy processed 25.60 billion subproblems, where the difference is merely 0.7%. On the other hand, the average speedup achieved by the O3S strategy is 750%, which dwarfs either 2.1% or 0.7%.

2.6.4 Evaluation of Data-Centric Strategy on Different Machines

We referenced a discussion about compute-centric and data-centric models in Section 2.1, and Section 2.6.3 demonstrated the necessity of shifting from a compute-centric (i.e., BFS)

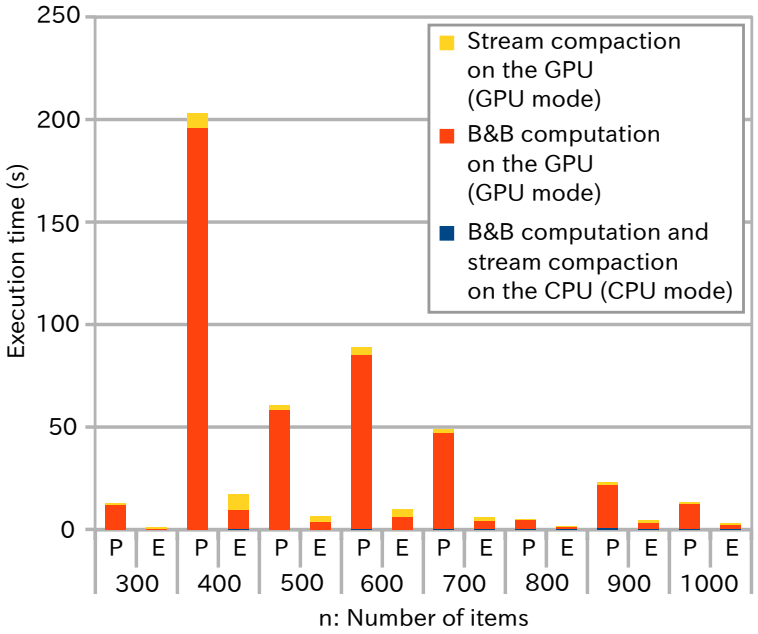

Figure 2.9: Comparison and breakdown of execution time for large instances that the in-core method failed to solve, due to memory exhaustion. [42] "P" and "E" refer to the previous [84] and extended methods, respectively. The O3S strategy outperformed the BFS strategy [84].

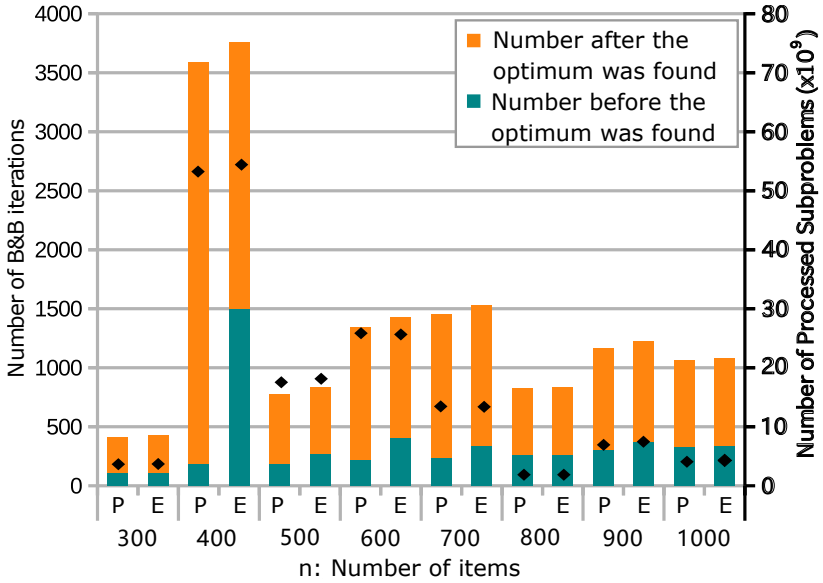

Figure 2.10: Comparison of the previous [84] and extended methods in terms of efficiency of finding the optimum when solving large instances. "P" and "E" refer to the previous and extended methods, respectively. Note that fewer iterations indicates better efficiency.

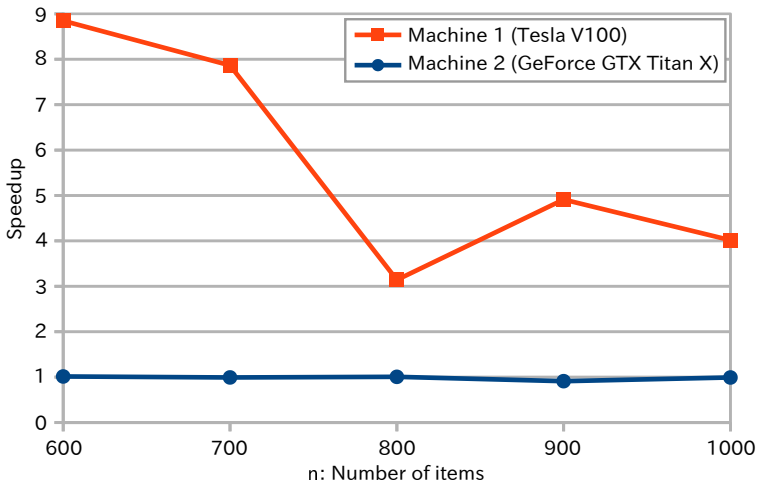

Figure 2.11: Speedup of data-centric strategy compared to compute-centric strategy on different machines.

strategy to a data-centric (i.e., O3S) strategy. However, it is difficult to conclude that a data-centric strategy is a one-size-fits-all solution. Therefore, we evaluated the data-centric strategy on Machine 1 (Tesla V100) and Machine 2 (GeForce GTX Titan X; testbed of the previous method [84]) in terms of execution time. Figure 2.11 shows different speedup values obtained by the extended method when solving instances with 600 to 1000 items on these two machines. Although speedup ranged from four to 8.8× was achieved by the data-centric strategy on Machine 1, there was no significant difference between the performance of the two strategies on Machine 2. As shown in Figure 2.2, the ratio of GPU memory bandwidth to PCIe bandwidth of Machine 2 is 21, whereas that of Machine 1 is 56, suggesting that Machine 2 did not have such huge difference between GPU memory and PCIe bandwidths as Machine 1. Therefore, the increased computation cost (i.e., more subproblems to be searched) in the data-centric strategy offset the reduced data transfer overhead.

Table 2.5 shows the profiling results of the two strategies when processing the instance of $n = 700$, which we use to verify our theoretical explanation for the aforementioned observations. Note that we use the bounding kernel to represent the GPU computation because it is the heaviest part (approximately 90%) of GPU B&B computation. The bounding kernel computes the upper and lower bounds based on two constant arrays (i.e., profits and weights of all the items). Because the two arrays are constant, they are preloaded in the read-only texture cache of each streaming multiprocesser (SM). Therefore, the read throughput of texture cache dominates the performance of the bounding kernel. As can be seen, the profiling results demonstrate that the huge improvement of memory access throughput and the constant, relatively low throughput of PCIe bus on the latest GPU make the data-centric strategy mandatory. On Machine 2 (GeForce Titan X), the texture read throughput is ap-

Table 2.5: Profiling results of the compute- and data-centric strategies when solving the instance of $n = 700$. The huge improvement of memory access throughput on Machine 1 (Tesla V100) necessitates the data-centric strategy. BET, TT1, TT2, TRT, PT1, and PT2 denote (1) average bounding execution time per iteration, (2) average CPU-to-GPU transfer time per iteration, (3) average GPU-to-CPU transfer time per iteration, (4) average texture cache read throughput, (5) average CPU-to-GPU PCIe throughput, and (6) average GPU-to-CPU PCIe throughput, respectively.

	Compute-centric strategy (BFS)						Data-centric strategy (O3S)					
Machine	BET (ms)	TT1 (ms)	TT2 (ms)	TRT (GB/s)	PT1 (GB/s)	PT2 (GB/s)	BET (ms)	TT1 (ms)	TT2 (ms)	TRT (GB/s)	PT1 (GB/s)	PT2 (GB/s)
Titan X	44.96	2.75	1.63	287.61	12.13	12.96	40.6	0.02	0.01	282.92	8.92	10.10
V100	2.32	2.72	1.62	5096.50	12.27	13.12	2.08	0.02	0.01	5073.60	8.78	10.71

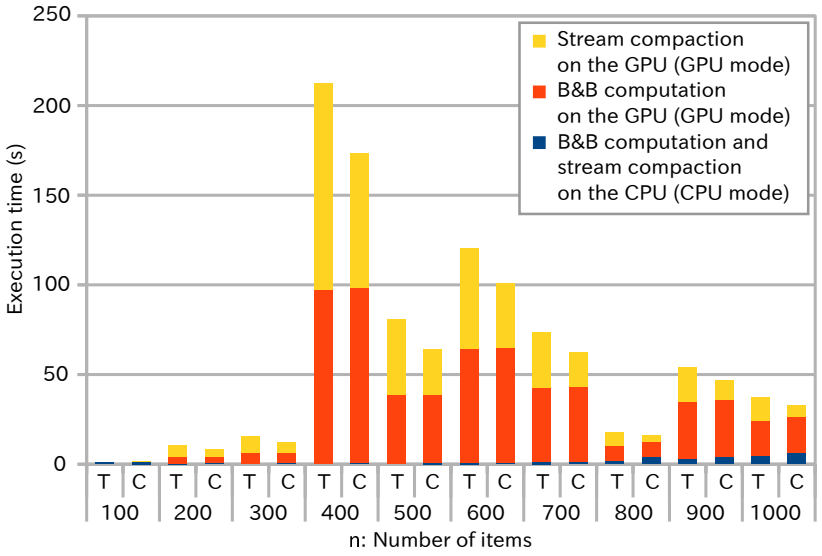

Figure 2.12: Comparison and breakdown of execution time of the Thrust and CUB libraries. "T" and "C" refer to the implementations with the Thrust and CUB libraries, respectively.

proximately 300 GBs, which makes the bounding kernel execution time long enough (more than 40 ms) to hide the CPU-GPU data transfer time no matter it is either 2 ms (compute-centric) or 0.02 ms (data-centric). However, on Machine 1 (Tesla V100), the texture read throughput is approximately 5100 GBs. Thus, the bounding kernel execution time is merely one twentieth of that on Machine 2. Therefore, the data transfer time cannot be hidden on Machine 1 if we still adopt the previous compute-centric strategy (BFS), due to the same PCIe throughput on both machines.

2.6.5 Comparison of Thrust and CUB Libraries

In the previous method [84], we used the Thrust library to realize the prefix sum computation required for the stream compaction strategy. Aiming at improving the stream compaction strategy, we compared the Thrust and CUB libraries in terms of execution time (Figure 2.12).

The results show that the stream compaction strategy realized using the CUB library was twice faster than that realized using the Thrust library, achieving an average speedup of 1.2× relative to total execution time. As mentioned earlier, the CUB library (1) has low-level and CUDA-specific optimizations and (2) allows us to improve our program by controlling the details of execution. Therefore, we achieved this speedup by implementing the GPU-based stream compaction with CUB.

2.7 Conclusion

In this paper, we have presented a GPU-accelerated, out-of-core B&B method for solving large 0-1 knapsack problems with adaptive CPU-GPU data transfer. The maximum problem size depends on the capacity of CPU memory rather than GPU memory. To realize this relaxation, the proposed method buffers promising subproblems in CPU memory rather than GPU memory. Because such a CPU-centric management scheme can suffer increased amount of data transfer between the CPU and GPU, the proposed method minimizes the number of subproblems to be transferred per iteration using an O3S strategy, condenses sparse data using a GPU-based stream compaction strategy, and hides data transfer overhead using an explicitly-managed pipelining strategy that overlaps data transfer with GPU-based B&B operations.

In our experiments, we found that the extended out-of-core method stored 41× as many subproblems at a time, solving approximately twice as many problems as a previous in-core method. In addition, for large instance classes, the extended out-of-core method also solved more instances than a previous out-of-core method because the O3S strategy suppressed subproblem splitting. We also found that our GPU-based stream compaction strategy was 9.9× as fast as a previous CPU-based stream compaction strategy. Furthermore, we found that the O3S strategy was 12.2× as fast as the previous BFS strategy on a Tesla V100 GPU, achieving a 7.5× speedup relative to total execution time, which paves the way to study solving the large 0-1 knapsack problem on the latest workstation GPUs with PCIe interconnects.

Additionally, we focused on strongly correlated 0-1 knapsack problems because such problems can generate massive subproblems to mandate out-of-core GPU computation. The number of solvable problems decreases with the increase of problem size, i.e., number of items. We avoided considering other cases such as weakly correlated problems because effective pruning would prevent generating enough subproblems to trigger GPU computation. However, the relationship between number of solvable problems and number of items for various kinds of knapsack problems is worth being studied, which remains future work.

In the future, we also plan to investigate more efficient upper bounds to improve efficiency of pruning subproblems and generalize the data-centric scheme to solve other problems on the GPU. Last but not least, we plan to combine the proposed O3S strategy with sophisticated subproblem-selection heuristics to further reduce the search time.

Chapter 3

A Data-Centric Directive-Based Framework to Accelerate Out-of-Core Stencil Computation on a GPU

3.1 Introduction

Stencil computation is one of the most important classes in scientific computing with a key principle of iteratively updating an input array by applying a fixed calculation pattern (i.e., *stencil*) to all the elements of the array. Stencil applications appear in a wide range of fields, including geophysics simulations [20, 80], computational electromagnetics [1], and image processing [31, 92]. Because computation time and memory consumption grow linearly with the size of input arrays, parallel implementations of stencil computation are of great importance [15]. Currently, the GPU is considered to be the most efficient architecture for parallel stencil code [78]. Armed with thousands of cores and 5–10 times as high memory bandwidth as CPUs, GPUs provide powerful solutions for both compute- and memory-intensive scientific problems [34, 55, 68, 85]. However, there are two main challenges in implementing GPU-accelerated stencil code: limited capacity of device (i.e., GPU) memory and considerable programming effort to implement GPU-accelerated code.

At several tens of GBs, the capacity of GPU memory is relatively small. Out-of-core methods are thus a straightforward option to process excess data. Although multi-node solutions are also used to handle large data, they are complex due to the need to reconcile intra- and inter-node programming models, such as Message Passing Interface (MPI) [24] and OpenMP [96]. Out-of-core methods decompose large data into smaller chunks such that each chunk fits in the GPU memory. As a result, out-of-core methods involve frequently moving data between the host (i.e., CPU) and device (i.e., GPU). Therefore, the programmer must deliberately organize data transfer to and from the GPU; otherwise, the performance of

parallel code is limited by data transfer.

Significant programming effort is required for the GPU-based parallelization of serial code, and the programming effort further increases if out-of-core computation needs to be implemented to handle excess data. Typically, CUDA [64] is used as a parallel programming framework for NVIDIA GPUs. To develop efficient CUDA programs, programmers must possess an in-depth knowledge of GPU-specific optimization techniques that adapt serial code and data structures to the highly parallel device [46]. To reduce the programming effort to develop GPU-based out-of-core code, directive-based programming frameworks such as OpenACC [70] have emerged as alternatives to CUDA. OpenACC provides users with a collection of compiler directives that can be simply inserted into the serial code. Such directives instruct the OpenACC compiler to automatically offload parallelizable workloads from the host to a parallel accelerator, such as a GPU. OpenACC is therefore gaining popularity among researchers because it helps port various scientific codes [5, 32, 83, 97, 105] to GPUs with less programming effort than low-level programming paradigms such as CUDA and OpenCL [41] do. However, due to the small GPU memory capacity, the simplicity of OpenACC (i.e., sharing of identical code and data structures between the CPU and GPU) can be problematic. For example, assume that we use a 100 GB array in a CPU-based code. If we naively insert OpenACC directives into this code, the OpenACC compiler will fail to prepare an array of the same size on the GPU due to GPU memory exhaustion.

Out-of-core computation is necessary to solve the aforementioned memory exhaustion problems. Miki *et al.* [54] proposed the pipelined accelerator (PACC), a directive-based framework that automatically generates out-of-core OpenACC code for large stencil applications. PACC decomposes large data into smaller chunks, overlaps data transfer with kernel execution, and performs *temporal blocking* to process on-GPU data for multiple times. However, the generated code failed to exploit the computational capabilities of state-of-the-art GPUs because the data transfer limits the performance, especially when the generated code was high-order stencil computation in which the calculation of an element relies on a wide neighboring area. Although the latest Tesla V100 GPUs have notably improved memory bandwidth (900 GB/s), CPU-GPU data transfer consumes significantly more time than kernel execution due to relatively slow interconnects (e.g., bandwidth of 16 GB/s for PCIe 3.0) and thus limits the performance. Moreover, the gap between GPU memory bandwidth and PCIe bandwidth widens over time [85]; therefore, we must focus more on reducing data transfer than improving computation. Thus, new data-centric computing techniques are needed to evolve the PACC framework on the latest GPUs.

In this paper, we extend the PACC framework [54] with two data-centric computing techniques (i.e., "schemes," used to distinguish Chapter 3 from Chapters 2 and 4) to alleviate

the performance limitation imposed by heavy data transfer.

The first is a direct-mapping scheme that eliminates CPU buffers [54] used to transfer decomposed data between the CPU and GPU. The second is a region-sharing scheme that significantly reduces CPU-GPU data transfer by further reusing on-GPU data. The experimental results demonstrate that even a relatively small dataset that fits in GPU memory benefits from the region-sharing scheme. This reveals the efficacy of region-sharing scheme regardless of the data size, as long as the data parallelism is sufficient for GPU acceleration.

The main contributions of this paper are as follows:

1. Extension of the PACC framework with two data-centric computing schemes that benefit large and high-order stencil applications.

2. Parallelization of a geophysics simulator [80]—a real world application—with the extended PACC framework.

The remainder of this paper is organized as follows. Section 3.2 introduces work related to GPU acceleration of out-of-core stencil computation, whereas Section 3.3 presents details of stencil computation with a real-world application—a geophysics simulator that was parallelized in this study. Section 3.4 introduces the PACC framework [54] that generates OpenACC-based out-of-core stencil code, whereas Section 3.5 elaborates on work that extends the PACC framework with data-centric computing schemes. Section 3.6 provides the experimental results, whereas Section 3.7 concludes the paper and provides suggestions for future work.

3.2 Related Work

Sourouri *et al.* [90] proposed a compiler framework for three-dimensional (3D) stencil computation on GPU clusters. They provided directives and a source-to-source translator and thus reduced programming effort by automatically generating out-of-core stencil code from serial code. However, techniques to reuse on-GPU data, such as temporal blocking, were not implemented.

Miki *et al.* [54] proposed PACC, an extension of OpenACC for out-of-core stencil computation on a GPU. The PACC framework automatically generates out-of-core code that decomposes the original data into chunks, each of which fits in the GPU memory. In addition, the generated code implements temporal blocking to reuse on-GPU data and performs pipeline execution to overlap data transfer with kernel execution. However, an intermediate-copying scheme is used in the generated code in which chunks are first copied to CPU buffers and then transferred to the GPU. Furthermore, the generated code transfers extra data (i.e.,

halo regions) to the GPU to perform temporal blocking. Therefore, the performance of the generated code degrades when the code handles high-order stencil computations and/or runs on the latest GPU due to the heavy data transfer cost. To address the data transfer problem, we integrated two data-centric computing schemes into the PACC framework. First, we adopted OpenACC APIs to map regions of the original data to the GPU buffers, avoiding data copying between the original data and CPU buffers. Secondly, we designed a region-sharing scheme for contiguous chunks to share common regions, thus significantly reducing the amount of data transfer between the CPU and GPU.

Jin *et al.* [36] proposed a multi-level optimization method for stencil computation on the domain that is bigger than memory capacity of GPU. In their method, they reused the intermediate computation results to eliminate redundant data (i.e., halo) transfers. Such a result-reusing scheme involves storing and restoring the intermediate results on the GPU for every two time steps, incurring more invocations for memory copy APIs than our region-sharing scheme. In our region-sharing scheme, one subdomain only needs to copy the overlapped regions for the consequent subdomain, before temporal blocking computations. The novelty of our work is that the region-sharing scheme is more succinct than the result-sharing scheme. That is, the region-sharing scheme has (1) fewer source lines and (2) simpler control that involves fewer invocations for CUDA APIs to copy data on the GPU. Although the region-sharing scheme is more succinct than the result-sharing scheme, the two schemes have almost the same performance because they have the same effect in reducing CPU-GPU data transfer that limits the performance. However, we must admit that compared to the region-sharing scheme, the result-reusing scheme reduces the size of GPU buffers by half the size of halo regions, because it can reuse the previous intermediate computation results in a shifting manner. The result-reusing scheme can also eliminate the redundant computations.

Shimokawabe *et al.* [87] proposed a stencil framework to realize large-scale computations beyond GPU memory capacity on GPU supercomputers. For excess data, the framework decomposes the entire domain stored in the CPU memory into subdomains and then transfers the subdomains to the GPU for computation. However, this method transfers each subdomain with all halos required for temporal blocking to the GPU. Therefore, the cost of the data transfer can be high, especially for high-order stencil code. In contrast, our method allows a subdomain to share common regions with contiguous subdomains to reduce the amount of data transfer.

Reguly *et al.* [74] presented a cache-blocking tiling technique that efficiently processes large-scale stencil code. Their implementations are based on OPS [56, 75, 76]—a domain specific language (DSL) that requires implementing stencil code based on its syntax. Instead, the PACC framework requires no modification aside from the insertion of directives into serial

```
1  //c0 = -205.0f/72.0f, c1 = 8.0f/5.0f,
2  //c2 = -1.0f/5.0f, c3 = 8.0f/315.0f,
3  //c4 = -1.0f/560.0f;
4  //p3, p2, and p1 are the pressures at time t+1, t,
5  //and t-1, respectively
6  p3[z][y][x] = (1/dx2 + 1/dy2 + 1/dz2) * c0 * p[z][y][x];
7  p3[z][y][x] += 1/dx2 * (c4 * (p2[z][y][x+4] + p2[z][y][x-4]) + c3 * (p2[z][y][x+3] + p2[z][y][x-3])
8                          + c2 * (p2[z][y][x+2] + p2[z][y][x-2]) + c1 * (p2[z][y][x+1] + p2[z][y][x-1]));
9  p3[z][y][x] += 1/dy2 * (c4 * (p2[z][y+4][x] + p2[z][y-4][x]) + c3 * (p2[z][y+3][x] + p2[z][y-3][x])
10                         + c2 * (p2[z][y+2][x] + p2[z][y-2][x]) + c1 * (p2[z][y+1][x] + p2[z][y-1][x]));
11 p3[z][y][x] += 1/dz2 * (c4 * (p2[z+4][y][x] + p2[z-4][y][x]) + c3 * (p2[z+3][y][x] + p2[z-3][y][x])
12                         + c2 * (p2[z+2][y][x] + p2[z-2][y][x]) + c1 * (p2[z+1][y][x] + p2[z-1][y][x]));
13 p3[z][y][x] = v[z][y][x] * v[z][y][x] * dt2 * p3[z][y][x] + 2 * p2[z][y][x] - p1[z][y][x];
```

Figure 3.1: 3D acoustic wave propagation described as 25-point stencil computation.

stencil code, thus reducing programming effort.

Hou *et al.* [29] proposed a framework to automatically generate stencil codes to access buffered data in the cached systems of GPUs. In their work, 2.5D block was adopted to decompose the original data; however, temporal blocking was not utilized for data reuse. In contrast, we adopted a 1.5D block scheme to decompose the original large data into blocks, fixing the sizes of blocks along the X and Y axes and varying only the size along the Z axis. We adopted the 1.5D block scheme for the convenience to apply temporal blocking because the scheme avoids discontinuous data intervened by halo regions along X and Y axes. Discontinuous data involve multiple invocations of CPU-GPU data transfer when the proposed region-sharing scheme is used to reduce transfers of halo regions.

Endo [18] proposed a recursive temporal blocking algorithm. Given multiple hierarchies of memory, he applied temporal blocking with variable block sizes. Data transfer and kernel execution did not overlap.

3.3 Acoustic Wave Propagator: Target Stencil Computation

In this section, we use the acoustic wave propagator [80] as an example of a reputed stencil application that is widely used for geophysics exploration in the petroleum industry. Acoustic wave propagation is governed by the following equation:

$$\frac{1}{V^2}\frac{\partial^2 p}{\partial t^2} = \nabla^2 p, \tag{3.1}$$

where $p(x, y, z, t)$ is the acoustic pressure at time t and the mesh specified by Cartesian coordination (x, y, z); $V(x, y, z)$ is the propagation speed, whereas ∇^2 is the Laplacian operator.

For our 3D application, $\nabla^2 p$ can be replaced by the sum of the second-order partial

```
1  #pragma pacc init
2
3  #pragma pacc pipeline targetinout(p1) targetin(p2) size([0:y][0:x]) halo([1:1][1:1]) async
4  for(int t=0;t<total_time steps;t++){
5    #pragma pacc loop dim(2)
6    for(int j=0;j<y;j++){
7        #pragma pacc loop dim(1)
8        for(int i=0;i<x;i++){
9            p2[j][i] = 0.25 * (p1[j][i+1] + p1[j][i-1] + p1[j+1][i] + p1[j-1][i]);
10       }
11   }
12   //swap p1 and p2
13   #pragma pacc loop dim(2)
14   for(int j=0;j<y;j++){
15       #pragma pacc loop dim(1)
16       for(int i=0;i<x;i++){
17           p1[j][i] = p2[j][i];
18       }
19   }
20 }
```

Figure 3.2: Sample code with PACC directives for five-point stencil computation, i.e., finite difference solver for Laplace's equation.

derivatives of p in each dimension. Therefore, we have

$$\frac{1}{V^2}\frac{\partial^2 p}{\partial t^2} = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2}. \tag{3.2}$$

For the partial derivatives, we use a second-order central finite difference approximation [21] for time and an eighth-order approximation for space. We thus have

$$\begin{aligned} \frac{1}{V^2}\frac{\partial^2 p}{\partial t^2} \approx \\ \frac{1}{V^2}\frac{p(x,y,z,t+1) - 2p(x,y,z,t) + p(x,y,z,t-1)}{dt^2}. \end{aligned} \tag{3.3}$$

With respect to the spatial terms on the right-hand side of (3.2), take the first term for instance, we have

$$\frac{\partial^2 p}{\partial x^2} \approx \frac{X}{dx^2}, \tag{3.4}$$

where

$$\begin{aligned} X = &-\frac{1}{560}(p(x+4,y,z,t) + p(x-4,y,z,t)) \\ &+\frac{8}{315}(p(x+3,y,z,t) + p(x-3,y,z,t)) \\ &-\frac{1}{5}(p(x+2,y,z,t) + p(x-2,y,z,t)) \\ &+\frac{8}{5}(p(x+1,y,z,t) + p(x-1,y,z,t)) \\ &-\frac{205}{72}p(x,y,z,t). \end{aligned} \tag{3.5}$$

```
allocate CPU datasets using cudaHostAlloc(h_p,h_sz*sizeof(float))
//p is pointer to original data, h_sz is size of data
allocate GPU buffers using d_buf[0-2]=acc_malloc(b_sz*sizeof(float))
//d_buf[0-2] are pointers to GPU buffers and b_sz is size of a GPU buffer
while number of remnant time steps is greater than zero
    reduce remnant time steps by temporal blocking time steps
    for each chunk
        map chunk to GPU buffer using acc_data_map(&h_p[ofs],d_buf[current],b_sz*sizeof(float))
        //ofs is offset from start of current chunk to start of original data
        #pragma data update device(h_p[ofs+r_sz:b_sz-r_sz])
        //r_sz is size of common regions
        copy common regions for next chunk
            using cudaMemcpyAsync(d_buf[current],d_buf[next]+ofs_b,r_sz*sizeof(float))
        //ofs_b is offset from start of GPU buffer to start of common regions
        for temporal blocking time steps
            #pragma acc kernels present(h_p[ofs:b_sz])
            parallel loops for stencil kernels
        #pragma data update host(h_p[ofs+r_sz/2:b_sz-r_sz])
        unmap data with acc_data_unmap(&h_p[ofs])
```

Figure 3.3: A simplified description showing how PACC-generated code utilizes OpenACC directives, OpenACC APIs and CUDA APIs.

The same procedure is then applied to the other two terms. In this way, we can discretize 3.1; that is, we can describe it in the form of a 25-point stencil computation (Figure 3.1) in which calculation of an element relies on 24 surrounding elements (i.e., halo regions).

3.4 PACC-Based Out-of-Core Stencil Computation

As explained in Section 3.1, to process excess data, naively inserting OpenACC directives into serial stencil code leads to GPU memory exhaustion. To avoid such failures, out-of-core stencil computation requires modifying the original code. A minimum modification includes data decomposition and CPU-GPU data transfer. In addition, techniques such as temporal blocking must be implemented to improve performance. To reduce the programming effort caused by these code modifications, the PACC framework [54] provides users with OpenACC-like directives to insert into their source code to specify the stencil computation regions that process excess data. The PACC framework then automatically translates the source code with PACC-directives to out-of-core OpenACC-based code. In this section, we describe how the framework realizes out-of-core stencil computation redin detail.

3.4.1 PACC Directives

The PACC framework provides OpenACC-like directives (i.e., *init*, *pipeline*, and *loop* constructs) to describe an out-of-core stencil code. Figure 3.2 presents an example of code with PACC directives for four-point stencil computation. Data decomposition and tempo-

ral blocking are not manually implemented but are automatically generated by the PACC framework. Brief descriptions for PACC directives are follows:

- The *init* construct (Line 1). This construct preserves variables used for out-of-core stencil computation such as the number of time steps for temporal blocking.
- The *pipeline* construct (Line 3). This construct is located directly before the outer loop which confines all iterations of the stencil computation. The *targetin* and *targetinout* clauses define read-only and read/write datasets. The *size* clause defines the size of datasets (i.e., range in each dimension) specified by the start index and length. The *halo* clause defines the size of halo regions in each dimension.
- The *loop* construct (Lines 5, 7, 13, and 15). This construct is located directly before each loop inside the kernel. The *dim* clause indicates the level of the loop inside the kernel. We decompose data at the top-level loop.

3.4.2 Rule-Based Translator

The PACC framework provides a translator that automatically generates out-of-core OpenACC code from a serial code implemented with PACC directives, based on rewriting rules. The translator was implemented with pure Python to achieve high usability by avoiding any package dependency issue. Code segments that perform data decomposition and temporal blocking are added by the translator. The translator functions as follows.

- Parses variables from the *pipeline* construct (e.g., the names of arrays and the size of halo regions).
- Adds variables and methods used to perform out-of-core computation (e.g., a method that maps data regions in the CPU memory to the GPU buffers).
- Replaces the *init* construct with code that initializes variables used to perform out-of-core computation (e.g., memory allocation for the GPU buffers).
- Adds an outer loop to the code block confined by the *pipeline* construct to control the chunk-wise out-of-core process.
- Rewrites all kernels in the code block confined by the *pipeline* construct (i.e., for each kernel, modifies the *loop* constructs and translates the data indices to process chunks rather than all of the data).

Figure 3.3 describes how the generated code is implemented with OpenACC directives, OpenACC APIs, and CUDA APIs.

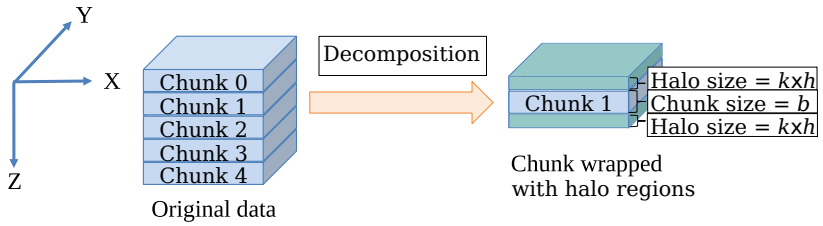

Figure 3.4: Data decomposition using 1.5D block scheme. The original data are decomposed into chunks along the main axis. Note that halo regions are required to be attached to each chunk for temporal blocking. In this figure, k and h denote the number of temporal blocking time steps and the size of halo regions, respectively.

3.4.3 Data Decomposition and Temporal Blocking

Because the OpenACC compiler prepares identical variables for both the host and the device, a large host array can cause runtime failures due to GPU memory exhaustion. Therefore, the PACC framework decomposes excess data into smaller chunks and move the chunks between the CPU and GPU for out-of-core computation. To the best of our knowledge, there are two explicit ways and an implicit way to perform the chunk-wise process:

1. intermediate-copying. In the previous PACC framework, Miki *et al.* [54] prepared CPU buffers. Therefore, chunks are copied from the original data to the CPU buffers and then transferred to the GPU buffers with the *update device* directive. Once computation on a chunk completes, the chunk is transferred back to the CPU buffers with the *update host* directive.

2. Direct-mapping. In this study, we extend the PACC framework with a direct-mapping scheme. This scheme directly maps a chunk from the original data to the GPU buffers with OpenACC map APIs. In this way, data copying between the original data and CPU buffers is eliminated.

3. Unified Memory [16]. This scheme relies on the CUDA runtime to stream data to and from the device, avoiding GPU memory exhaustion. However, this scheme has two disadvantages that significantly impair the performance of out-of-core code. First, the scheme is unaware of the application-specific optimizations, such as temporal blocking. Secondly, the scheme results in low CPU-GPU bandwidths due to a complex page fault handling mechanism if prefetching hints are not explicitly specified [60].

In most cases, stencil code is time-evolving, which signifies that all of the data must be calculated for multiple time steps; therefore, temporal blocking must be implemented to reduce the amount of CPU-GPU data transfer. As mentioned in Section 3.2, we decompose

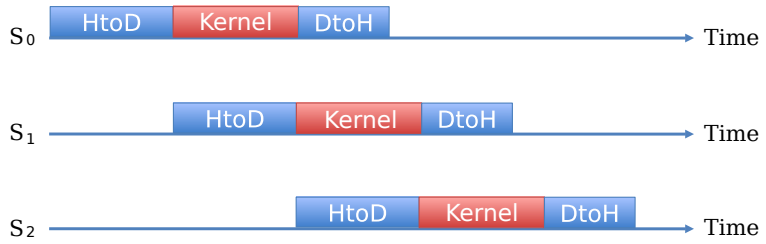

Figure 3.5: Asynchronous CUDA streams used to overlap data transfer with kernel execution. S_0, S_1, and S_2 are streams used for the i-th, $(i+1)$-th, and $(i+2)$-th chunks, respectively. Bars marked as "HtoD," "DtoH," and "Kernel" represent CPU-to-GPU data transfer, GPU-to-CPU data transfer, and kernel execution, respectively.

the original data using a 1.5D block scheme for the convenience to implement temporal blocking. Note that 2D and 3D decomposition schemes are not considered for two reasons. First, although 2D and 3D decomposition schemes can reduce the size of chunks and thus reduce the data transfer, on the other hand, the amount of data that can be reused on the GPU is also reduced, which is unfavorable. Secondly, 2D and 3D decomposition schemes make a chunk non-contiguous in the memory, impairing the efficiency in transferring the chunks between the CPU and GPU. Precisely, either multiple invocations of data-transfer API or buffers used to combine non-contiguous data before transfer is needed for each chunk because the chunk consists of multiple non-contiguous data segments. For high usability, the previous PACC framework implemented temporal blocking with a succinct overlapped tiling scheme. This scheme simply wraps each chunk with halo regions (Figure 3.4) and thus avoids analyzing data dependencies across multiple arrays, which must be performed if temporal blocking is implemented using wavefront [52, 57, 75, 103], trapezoidal [25], and diamond [6] tiling schemes.

However, halo transfer increases linearly with the number of time steps to update a chunk on the GPU. For high-order stencil code that has a large halo area, transferring extra data can be incredibly time consuming. To address this problem, we improved the performance of the overlapped tiling scheme with a data reuse scheme that significantly reduces CPU-to-GPU (HtoD) data transfer, which is described in greater detail in Section 3.5.2.

3.4.4 Pipeline Execution

At runtime, the process of a chunk has three stages: (1) HtoD transfer, (2) kernel execution, and (3) GPU-to-CPU (DtoH) transfer. The previous PACC framework [54] thus adopted a three-stage pipeline to process three chunks in parallel with asynchronous CUDA streams. For each chunk, the framework submits operations to the corresponding stream (Figure 3.5). In this way. data transfer is overlapped with kernel execution.

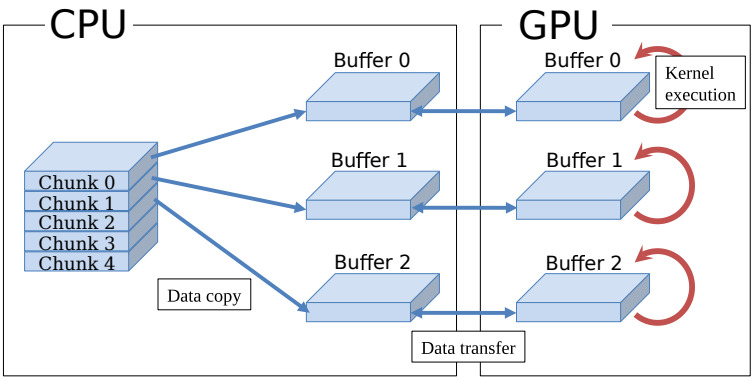

Figure 3.6: intermediate-copying scheme [54]. CPU buffers are used as "transfer stations" between original data and GPU buffers.

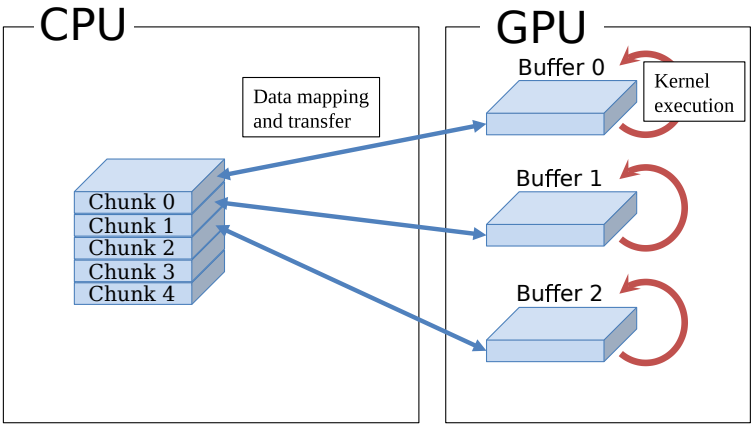

Figure 3.7: Direct-mapping scheme implemented with data mapping APIs provided by OpenACC. This scheme eliminates data copying between the original data and CPU buffers.

3.5 Data-Centric Computing Schemes

As mentioned in Section 3.4.3, data transfer limits the performance of high-order stencil code generated by the PACC framework [54], because a large amount of halo data is transferred between the CPU and GPU at runtime. This limitation is more noticeable on new GPUs due to a widened gap between GPU memory bandwidth and PCIe bandwidth [85].

To alleviate this limitation, we extend PACC with two data-centric computing schemes—direct-mapping and region-sharing—that notably reduce the amount of data transfer. The direct-mapping scheme maps and transfers regions of the original data to the GPU buffers, avoiding data copying between the original data and CPU buffers. Furthermore, the region-sharing scheme evolves the overlapped tiling scheme used by the previous PACC framework by eliminating all halo transfer between the CPU and GPU.

3.5.1 Direct-Mapping Scheme

The previous PACC framework uses an intermediate-copying scheme that prepares CPU buffers that mediate between the original data and GPU buffers (Figure 3.6). In contrast, the proposed direct-mapping scheme uses OpenACC APIs (i.e., *acc_data_map* and *acc_data_unmap*) to map regions of the original data to the GPU buffers (Figure 3.7). The mapped data regions are then transferred between the CPU and GPU with OpenACC directives (i.e., *data update device* and *data update host*). Thus, the direct-mapping scheme avoids data copying between the original data and CPU buffers.

Nevertheless, we take advantage of page-locked (pinned) memory for high speed of data transfer between CPU and GPU. Similar to the CUDA implementations [2, 48, 99], the direct-mapping uses pinned memory by allocating the CPU datasets with CUDA API, but it differs from [2, 48, 99] in using OpenACC APIs to map host data to GPU buffers (Figure 3.3). Moreover, because the generated code deploys OpenACC directives, we compiled the generated code with the PGI compiler option to enable pinned memory. Without the appropriate option, the direct-mapping uses pageable memory. Note also that the direct-mapping scheme consumes more pinned memory than the intermediate-copying scheme does. In the intermediate-copying scheme, only three CPU buffers, each with a size of a chunk and corresponding halo regions, need to be allocated with pinned memory. However, in the direct-mapping scheme, all of the original data must be allocated with pinned memory. This drawback thus requires dynamically allocating pinned memory if the size of the original data is beyond the CPU memory capacity.

3.5.2 Region-Sharing Scheme

The previous PACC framework performs temporal blocking with the overlapped tiling scheme to avoid analyzing data dependencies across multiple arrays. However, as we mentioned in Section 3.4.3, the overlapped tiling scheme requires attaching extra data (i.e., halos) to each chunk. The extra data transfer increases with the number of time steps of temporal blocking and is time-consuming for high-order stencil code due to large halo areas. We therefore propose a region-sharing scheme to eliminate the extra data transfer, which constitutes a large proportion of the HtoD data transfer.

The region-sharing scheme takes advantage of the fact that two contiguous chunks share common regions (Figure 3.8). We assume that a stencil has equal sizes of halo regions in up and down directions, which is the most common case in stencil computation. As a result, the region-sharing scheme can copy the common regions of a chunk that has arrived on the GPU to the GPU buffers that would store the next chunk. The scheme then only needs

Algorithm 2: Performing out-of-core stencil computation for T time steps using the region-sharing scheme. Note that we assume that the sizes of top halo H_i^{top} and bottom halo H_i^{bottom} for any chunk C_i are equal. For simplicity, we avoid special descriptions for uneven top and bottom halo regions such as that of the first chunk.

Input: (1) $\bigcup_{i=0}^{n-1} C_i$: original data, which is decomposed into n chunks, (2)S_0, S_1, S_2: CUDA streams to perform HtoD data transfer and common regions copying for three chunks, (3)S_3, S_4, S_5: CUDA streams to perform kernel execution and DtoH data transfer for three chunks, (4) T and K: number of total time steps and number of time steps for temporal blocking, respectively.

Output: $\bigcup_{i=0}^{n} C_i$: data that has been updated for T time steps.

```
1  while T > 0 do
2      k ⇐ min(T, K)
3      T ⇐ T − k
4      i ⇐ 0
5      while i < n do
6          sid ⇐ i (mod 3)                          // id of first stream for current chunk
7          prev ⇐ sid − 1                           // id of first stream for previous chunk
8          if prev ≠ −1 then
9              transfer C_{i−1} to CPU using S_{prev+3}  // transfer updated previous chunk to
                CPU
10         if i = 0 then
                   // for first chunk, transfer it with upper and lower halo regions
11             transfer H_i^top ∪ C_i ∪ H_i^bottom to GPU using S_sid
12         else
                  // for each of other chunks, transfer remnant of it with lower halo
                   regions
13             transfer (C_i ∪ H_i^bottom − H_{i−1}^bottom) to GPU using S_sid
14         if prev ≠ −1 then
15             wait for S_prev to complete copying common regions
16         if i ≠ n − 1 then
               // if current chunk is not last one, copy common regions to next chunk
17             copy H_{i+1}^top ∪ H_i^bottom for C_{i+1} on GPU using S_sid
18         wait for S_sid to complete HtoD data transfer
19         while k > 0 do
20             process C_i on GPU using S_{sid+3}
21             k ⇐ k − 1
22         i ⇐ i + 1
```

to transfer the remnant of the next chunk with corresponding halo regions. Note that the

Algorithm 3: Performing out-of-core stencil computation for T time steps using the result-reusing scheme [36].

Input: (1) $\bigcup_{i=0}^{n-1} C_i$: original data, which is decomposed into n chunks, (2)S_0, S_1, S_2: CUDA streams to perform HtoD and DtoH data transfers for three chunks, (3)S_3, S_4, S_5: CUDA streams to perform reading and writing intermediate results and kernel execution for three chunks, (4) T and K: number of total time steps and number of time steps for temporal blocking, respectively.

Output: $\bigcup_{i=0}^{n} C_i$: data that has been updated for T time steps.

```
1  while T > 0 do
2      k ⇐ min(T, K)
3      T ⇐ T − k
4      i ⇐ 0
5      while i < n do
6          sid ⇐ i (mod 3)                          // id of first stream for current chunk
7          prev ⇐ sid − 1                           // id of first stream for previous chunk
8          if prev ≠ −1 then
                                                    // transfer updated previous chunk to CPU
9              transfer C_{i−1} to CPU using S_prev
10         if i = 0 then
                   // for first chunk, transfer it with upper and lower halo regions
11             transfer H_i^top ∪ C_i ∪ H_i^bottom to GPU using S_sid
12         else
                 // for each of other chunks, transfer remnant of it with lower halo
               regions
13             transfer (C_i ∪ H_i^bottom − H_{i−1}^bottom) to GPU using S_sid
14         wait for S_sid to complete HtoD data transfer
15         if prev ≠ −1 then
16             wait for S_{prev+3} to complete writing intermediate results
17         while k > 0 do
18             if i ≠ 0 and k (mod 2) = 0 then
19                 read the on-GPU intermediate results of the common regions
                     computed by C_{i−1}
20                 write the intermediate results of the common regions on GPU for C_{i+1}
21             process C_i on GPU using S_{sid+3}
22             k ⇐ k − 1
23         i ⇐ i + 1
```

size of the remnant equals that of a chunk without halo regions; thus, the scheme effectively eliminates all halo transfer between the CPU and GPU.

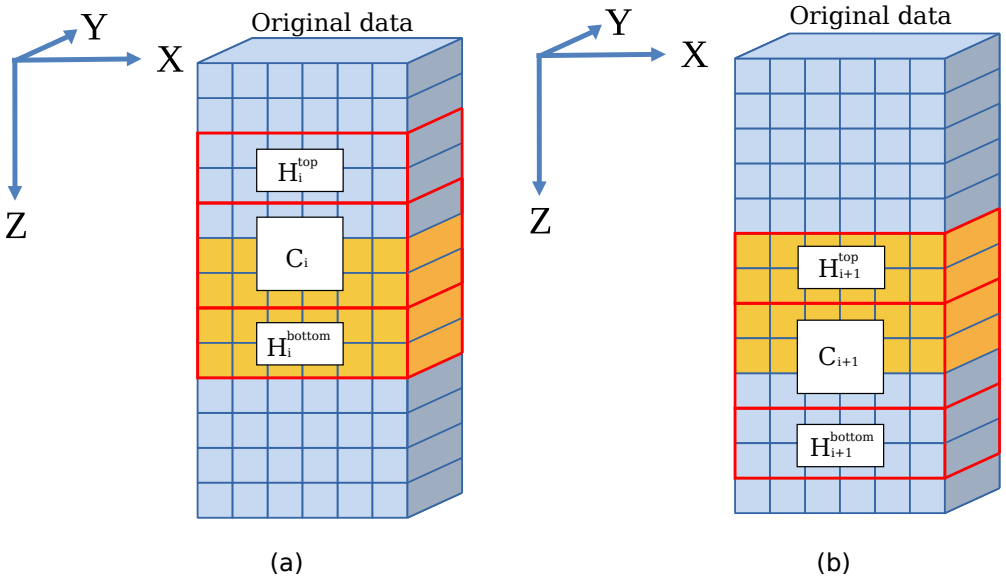

Figure 3.8: Regions shared by two contiguous chunks (a) C_i and (b) C_{i+1}, $H_{i+1}^{top} \cup H_i^{bottom}$. H_{i+1}^{top} denotes the top halo regions of C_{i+1}, whereas H_i^{bottom} denotes the bottom halo regions of C_i. Note that the original data are decomposed along the Z axis, and each chunk therefore has top and bottom halo regions. Chunks and halo regions are outlined in red, whereas common regions are colored yellow.

Theoretically, the more halo regions are used, the more benefit can be drawn from the region-sharing scheme. That is, more time can be saved that would otherwise be consumed by data transfer. However, in practice, more halo regions require larger GPU buffers; thus, the region-sharing scheme is also subject to limitations of GPU memory capacity.

The region-sharing scheme requires (1) copying common regions between chunks on the GPU and (2) maintaining an inter-chunk execution order (i.e., computation on a chunk must wait for the previous chunk to complete copying common regions). To fulfill the first requirement, we use *cudaMemcpyAsync* to perform data copying on the GPU. To fulfill the second requirement, we control the pipeline execution in smaller granularity than that used in the previous PACC framework.

Algorithm 2 provides the details of out-of-core stencil computation using the region-sharing scheme. We assume that upper and lower halo regions have the same size. Six streams are used for three chunks. Specifically, we use two streams to perform operations for a chunk. The first stream transfers the chunk from the CPU to the GPU and copies the common regions of the chunk to the next chunk. The second stream computes on the chunk and transfers the computation results back to the CPU. Note that the inter-chunk execution order requires extra synchronizations (Line 16 and 17). For instance, considering three contiguous chunks, C_0, C_1, and C_2, C_1 must wait for C_0 to complete copying common regions before C_1 can copy common regions to C_2.

Algorithm 3 shows the details of a result-reusing scheme [36] mentioned in Section 3.2. In the temporal blocking loops of this scheme, for every two time steps, a chunk reads the intermediate results of common regions computed by the previous chunk (line 19), and writes the intermediate results of common regions computed by itself for the next chunk (lines 20). In contrast, the region-sharing scheme is more succinct (i.e., having fewer lines of source code) and reduces the invocations of APIs to perform on-GPU copy, which is because the region-sharing scheme avoids maintaining additional GPU buffers to store intermediate results and involves copying the common regions only once before the temporal blocking loops. Moreover, the two schemes have the same effect in eliminating halo transfers, resulting in almost the same performance achievement, which will be demonstrated with experimental results in Section 3.6.5.

However, the region-sharing scheme has a disadvantage. If the code runs in a multi-node environment, the inter-chunk execution order prevents the assignment of chunks to different nodes in an arbitrary order because contiguous chunks must be assigned to the same node to share common regions. This constraint reduces the flexibility with which load balancing is performed in a multi-node environment. Nevertheless, the performance improvement produced by the region-sharing scheme outweighs the reduction in programming flexibility.

3.6 Experimental Results

We parallelized an acoustic wave propagator introduced in Section 3.3 using the extended PACC framework that automatically generated out-of-core OpenACC-based code from the original serial code. To demonstrate the applicability to other stencil kernels, we also applied the extended PACC framework to the Himeno benchmark [77], which is widely used in measuring computation speed of different architectures. Himeno benchmark has much smaller halo size (i.e., one) than the acoustic wave propagator. The generated code implemented the proposed schemes, and we evaluated the generated code on a latest NVIDIA GPU with respect to performance.

3.6.1 Experimental Setup

To verify the effectiveness of the proposed schemes on a latest NVIDIA GPU (Table 3.1), we designed five experiments with two datasets for the acoustic wave propagator (Table 3.2) and one dataset for the Himeno benchmark (Table 3.3). A brief introduction to the four experiments is as follows.

- The first experiment aims to detect the performance improvement achieved by the

Table 3.1: Testbed: a latest NVIDIA GPU.

GPU	Tesla V100-PCIe
GPU architecture	Volta
GPU memory capacity	16 GB
CPU	Xeon Silver 4110
OS	Ubuntu 16.04.6
CUDA	9.2
PGI compiler	19.10
Interconnect	PCIe 3.0 ×16

Table 3.2: Two datasets for acoustic wave propagator.

	Out-of-core	In-core
Size	$1160 \times 1160 \times 1160$	$820 \times 820 \times 820$
Data type	Float	Float
Number of arrays	4	4
Total amount	24 GB	8 GB

Table 3.3: Dataset for Himeno benchmark.

Size	$613 \times 613 \times 1225$
Data type	Float
Number of arrays	14
Total amount	24 GB

PACC-generated code compared with other implementations, such as OpenMP and Unified Memory based programs.

- The second experiment aims to analyze the speedups achieved by the direct-mapping and region-sharing schemes.

- The third experiment aims to investigate the improvement (or degradation) attributed to PACC-generated out-of-core code compared with an in-core implementation.

- The fourth experiment aims to compare the region-sharing scheme with a previous result-reusing scheme in terms of performance.

- The fifth experiment aims to demonstrate the number of source lines automatically generated by the PACC framework.

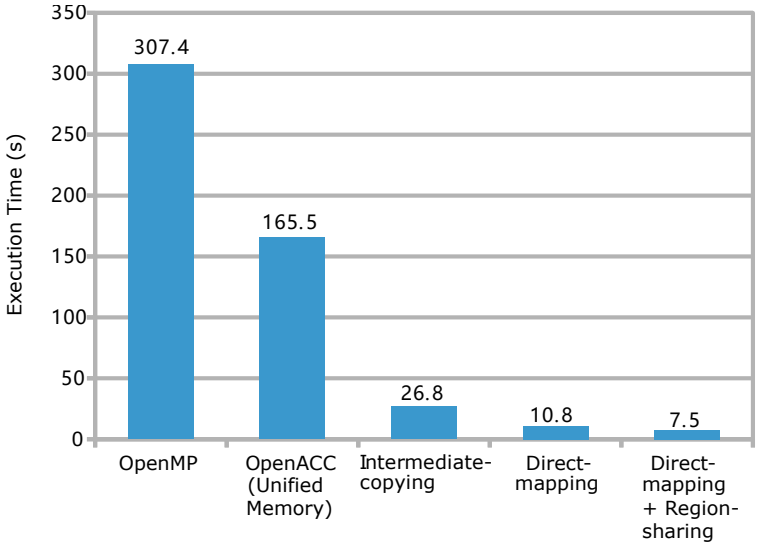

Figure 3.9: Comparison of the acoustic wave propagation code generated by the extended PACC framework (Direct-mapping + Region-sharing) with other implementations on a Tesla V100 GPU in terms of performance.

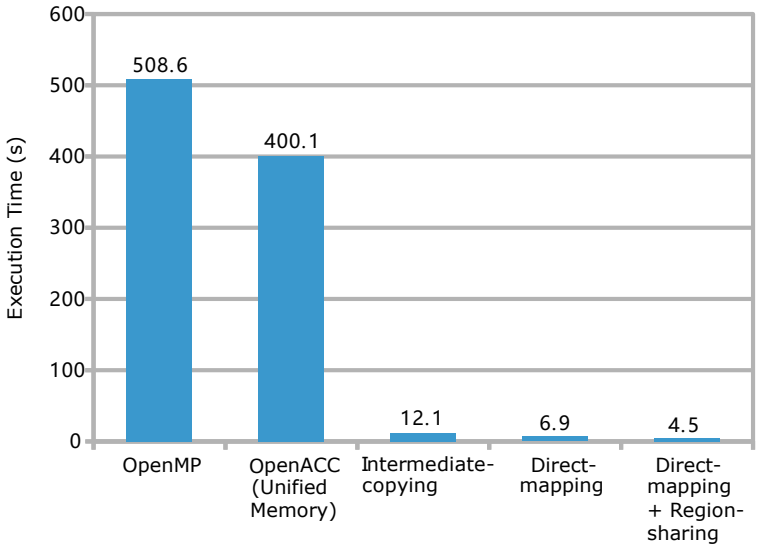

Figure 3.10: Comparison of the Himeno benchmark code generated by the extended PACC framework (Direct-mapping + Region-sharing) with other implementations on a Tesla V100 GPU in terms of performance.

3.6.2 Comparison with OpenMP, Unified Memory, and Intermediate-Copying Based Implementations

In this experiment, we compared the out-of-core acoustic wave propagation code and Himeno benchmark generated by the extended PACC framework with OpenMP (i.e., CPU parallelization), Unified Memory, intermediate-copying [54], and direct-mapping based im-

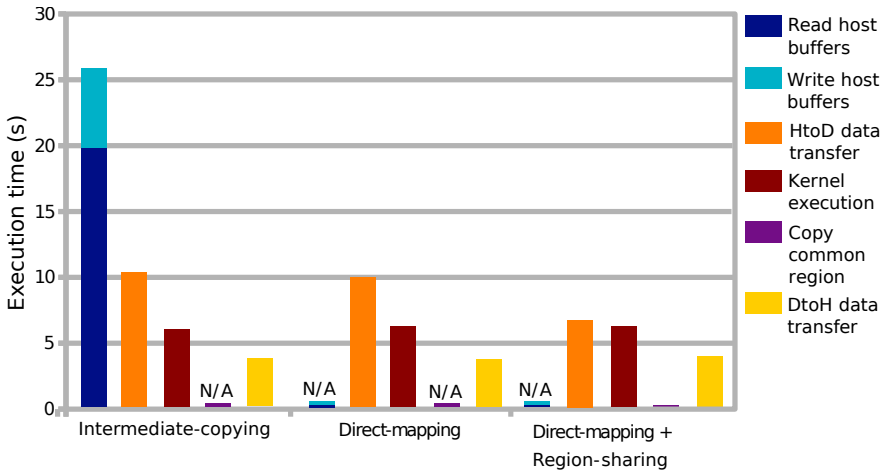

Figure 3.11: Comparison of the breakdown of the execution time of three versions of out-of-core code, implemented with intermediate-copying [54], direct mapping, and both direct mapping and region sharing, respectively. Bars of different colors denote the time consumed by different operations. "N/A" denotes that no time consumption for that operation. Direct-mapping based code eliminated data copying between the original data and CPU buffers, and thus ran 2.7× as fast as intermediate-copying based code; moreover, region-sharing achieved a further 1.4× speedup because it eliminated all halo transfer.

plementations. Note that in our experiments, we prepared the OpenMP and OpenACC based implementations by inserting OpenMP or OpenACC directives into the original serial code. Unified Memory was enabled for the OpenACC-based implementation by compiling the source code with the Unified Memory option. Techniques, such as loop transformation and Unified Memory prefetching hints, were not used because they require many manual modifications to the original code, such as dividing the original data into tiles. In contrast, PACC-based out-of-core implementations were automatically generated by the proposed framework. All PACC-based implementations used pinned memory by enabling pinned memory when we compiled the codes. We set d (the number of chunks) to eight, and k (the number of temporal blocking time steps) to 12 for the acoustic wave propagator, whereas we set d to 10, and k to 16 for the Himeno benchmark. These feasible settings were experimentally determined. The datasets used in this experiment were 24 GB (referring to out-of-core data in Table 3.2 and Table 3.3).

Figure 3.9 presents the results of running acoustic wave propagation codes obtained on the Tesla V100 GPU. The execution time of out-of-core code generated by the extended PACC framework was 7.5 s, which was 41.0×, 22.1×, 3.6×, and 1.4× as fast as the OpenMP (307.4 s), Unified Memory (165.5 s), intermediate-copying [54] (26.8 s), and direct-mapping (10.8 s) based implementations, respectively.

Figure 3.10 demonstrates that the extended PACC framework obtained even better speedup for the Himeno benchmark than for the acoustic wave propagator on the Tesla V100

GPU. The execution time of out-of-core code generated by the extended PACC framework was 4.5 s, which was 103.0×, 88.9×, 2.7×, and 1.5× as fast as the OpenMP (508.6 s), Unified Memory (400.1 s), intermediate-copying [54] (12.1 s), and direct-mapping (6.9 s) based implementations, respectively. Note that the transition of speedups was similar for both acoustic wave propagator and Himeno benchmark, as we incrementally added the schemes to the PACC framework, which proved the general usefulness of the extended PACC framework for different stencil kernels.

These results demonstrate the effectiveness of the extended PACC framework on the state-of-the-art GPU, owing to the direct-mapping and region-sharing schemes. Detailed analyses for the improvements achieved by the two schemes are given in the following section.

3.6.3 Detailed Analyses of Data-Centric Computing Schemes

To discuss the performance bottleneck and achievements of the data-centric computing schemes, we analyzed the performance improvement by using the direct-mapping and region-sharing schemes. In this experiment, we ran the intermediate-copying, direct-mapping, and direct-mapping plus region-sharing based implementations on the Tesla V100 GPU. Because the extended PACC framework obtained similar benefits for both acoustic wave propagtor and Himeno benchmark, we considered only the acoustic wave propagtor in this experiment.

Figure 3.11 reveals that the replacement of the intermediate-copying scheme by the direct-mapping scheme attained a 2.7× speedup (26.8 s/10.1 s). The performance (i.e., total execution time) of intermediate-copying based code was apparently limited by the data copying between the original data and CPU buffers (i.e., reading and writing the CPU buffers). Note that in the intermediate-copying [54] based code, although reading and writing the CPU buffers were performed by multiple threads, reading was not overlapped with writing, and vice versa. In contrast, the performance of the direct-mapping based code was limited by the HtoD data transfer, which required less time than reading and writing the CPU buffers did. The amount of HtoD data transfer was the same for both schemes; thus, a reduction in data transfer is necessary for further improvement.

Moreover, for the code implemented with direct-mapping and that implemented with both direct-mapping and region-sharing, the bounding operation is HtoD data transfer. Therefore, we are interested in analyzing the theoretical and practical speedups for HtoD data transfer achieved by the region-sharing scheme. We obtained the theoretical speedup by calculating the amount of HtoD data transfer reduced by the region-sharing scheme. In this experiment, the data consisted of four arrays. Each array had $1160 \times 1160 \times 1160$ float-type elements. Because we decomposed the data along one axis, we considered the data as 1160 planes, where each plane had 1160×1160 float-type elements. We decomposed the data into

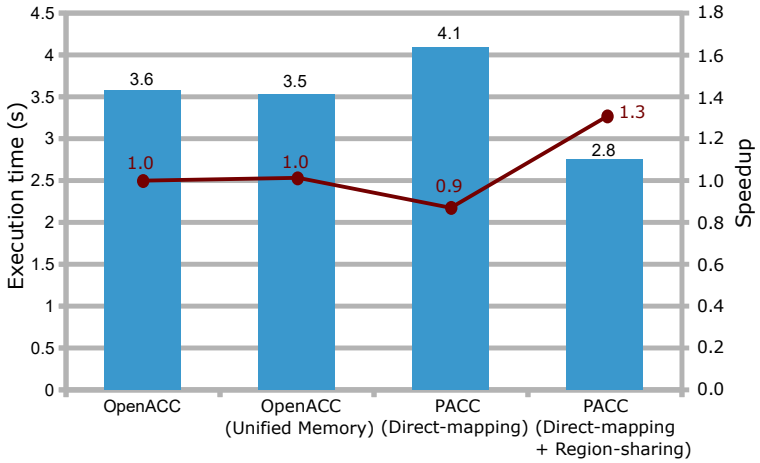

Figure 3.12: Analysis of the performance of PACC-generated code to process data that fit in the GPU memory.

eight chunks and processed each chunk on the GPU for 12 times. Therefore, for the first and last chunks, the halo region had $4 + 4 \times 12 = 52$ planes, whereas for the other chunks, the halo regions had $2 \times 4 \times 12 = 96$ planes. Because the region-sharing scheme effectively avoids transferring halo regions, the amount of HtoD transfer equals that of the original data: 1160 planes. However, if the region-sharing scheme is not used, $(1160 + 52 \times 2 + 96 \times 6 = 1,840$ planes must be transferred to the GPU to process all of the data for 12 time steps. Summarily, the theoretical speedup of HtoD transfer should be $1,840/1160 = 1.6\times$. However, the practical speedup for HtoD transfer was $1.5\times$ (=9.8 s/6.7 s). In order to explain the discrepancy, we recorded the data transfer behaviors of both implementations. The results indicate that the increased number of operation streams and synchronizations required by the region-sharing scheme resulted in a small achieved bandwidth compared with the implementation without region-sharing. For the HtoD data transfer, the implementation with region-sharing achieved 10.5 GB/s, whereas the implementation without region-sharing achieved 11.2 GB/s. We can thus verify the validity of the practical speedup, i.e., $(1,840/11.2)/(1160/10.5) = 1.5$ times. In terms of the total execution time, the code implemented both direct-mapping and region-sharing (7.5 s) ran $1.4\times$ as fast as that using direct-mapping alone (10.1 s).

3.6.4 Comparison with In-Core Implementation

In this experiment, we aimed to determine the improvement or degradation caused by PACC-generated out-of-core code, compared with in-core OpenACC code. We ran OpenACC, Unified Memory, direct-mapping, and direct-mapping plus region-sharing based implementations on the Tesla V100 GPU. The direct-mapping plus region-sharing based implementation implemented the two data-centric computing schemes to process data that fit in the GPU

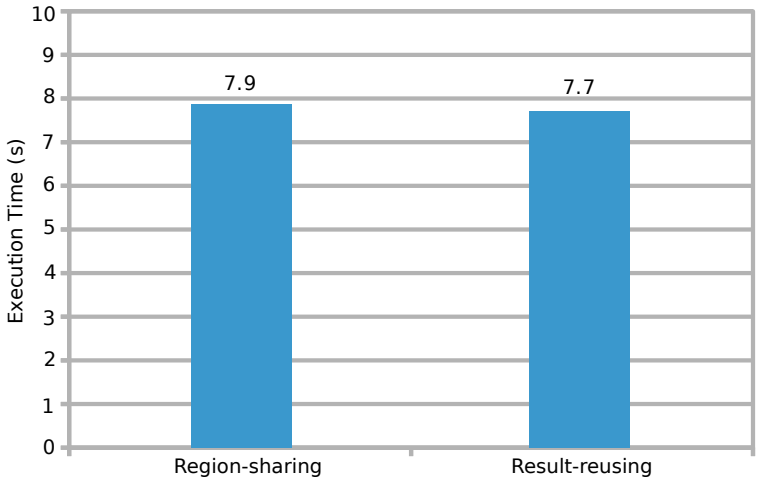

Figure 3.13: Comparison between region-sharing and result-reusing [36] schemes in terms of performance.

Table 3.4: Comparison between source lines of code (SLOC) of PACC-generated out-of-core code (429 lines) and that of serial code (185 lines).

	Serial code	Code with PACC directives	Generated out-of-core code
SLOC	185	193	429

Table 3.5: Details of generated out-of-core code. "Changed" and "Unchanged" denote lines that are changed and not changed, respectively, compared with serial code.

	Changed				Unchanged
	Adding new variables and methods	Data transfer (direct mapping)	Region sharing	Kernel execution (temporal blocking)	
SLOC	78	132	25	90	104

memory (in-core data in Table 3.2). The direct-mapping based implementation processed the same data.

Figure 3.12 reveals that the Unified Memory based code had almost the same performance as OpenACC based code without using Unified Memory, implying that the Unified Memory technique fails to sufficiently overlap data transfer with kernel execution if prefetching hints are not explicitly implemented. The PACC-generated code using the direct-mapping scheme alone led to a degradation of 15% compared with the in-core code due to extra data (i.e., halo regions) transfer. Fortunately, the code using both the direct-mapping and region-sharing schemes ran 1.3× as fast as the in-core code, because the proposed schemes notably reduce the amount of data transfer and thus improve the effect of overlapping data transfer with kernel execution.

3.6.5 Comparison with Result-Reusing Scheme

In this experiment, we compared the region-sharing scheme with result-reusing scheme [36]. Both implementations used direct-mapping scheme.

Figure 3.13 shows that similar performances were obtained by the two schemes, which was because the two schemes had the same effect in eliminating the halo transfers. The performances were limited by HtoD data transfer time for both schemes. Precisely, the result-reusing scheme had shorter HtoD data transfer time (6.9 s) than region-sharing scheme (7.3 s), because the result-sharing scheme took advantage of smaller GPU buffers. However, the region-sharing scheme has shorter on-GPU data copy (i.e., common region copy) time (0.1 s) than result-reusing scheme (0.2 s), and the region-sharing scheme also showed better effect in overlapping operations, because the region-sharing scheme had fewer CUDA API invocations for on-GPU data copy.

3.6.6 Evaluation of Programming Effort Benefits

In this experiment, we examined on the programming effort reduced by using the extended PACC framework. We considered the source lines of code (SLOC) as a metric for programming effort. Table 3.4 demonstrates that the serial code of the acoustic wave propagator had 193 lines, whereas the PACC-generated out-of-core code had 429 lines. Therefore, the PACC code generation process reduced the programming effort by automatically extending the serial code 2.3 times in length. In fact, the framework was even more helpful because 325 out of 429 generated lines were either new or modified compared with the original serial code (Table 3.5). From this perspective, the PACC framework exempts the users from manually modifying 75% of the final program.

3.7 Conclusion

In this study, we extended the PACC framework using two data-centric computing schemes for GPU acceleration of out-of-core stencil computation. The direct-mapping scheme avoids data copying between the original data and CPU buffers, whereas the region-sharing scheme avoids halo region transfer between the CPU and GPU. We thus retain the high usability of this directive-based framework and generate efficient out-of-core code with the framework.

The experimental results demonstrate that out-of-core code generated by the extended PACC framework outperformed OpenMP and Unified Memory based implementations by a factor of 10 on a latest GPU, thus verifying the usefulness of the extended PACC framework. With respect to the data-centric computing schemes, the replacement of intermediate-

copying [54] by the direct-mapping scheme contributed to a 2.7× speedup. Furthermore, the region-sharing scheme achieved an additional 1.4× speedup. Although being aimed at out-of-core stencil computation, the extended PACC framework can be applied to in-core code, achieving a 1.3× speedup. With respect to the extent of programming effort reduction, we determined that the PACC framework automatically extended the original serial code 2.3 times in length to obtain the out-of-core parallel code. In addition, 75% of the extended code was different from the original serial code.

Future work includes adapting the PACC framework to a multi-node environment. Because the region-sharing scheme involves an inter-chunk execution order, sophisticated offloading algorithms are required that consider the data dependency between contiguous chunks. Tuning schemes are also worthy of investigation to automatically determine the parameters for temporal blocking.

Chapter 4

Accelerating Out-of-Core GPU Stencil Computation with On-the-Fly Compression

4.1 Introduction

Stencil computation is the backbone of many scientific applications, such as geophysics simulations [20, 80], computational electromagnetics [1], and image processing [92]. The key principle of stencil computation is to iteratively apply a fixed calculation pattern (stencil) to every element of the input datasets. Such a SIMD characteristic of stencil computation makes itself a perfect scenario to use GPUs for acceleration. A GPU has thousands of cores and the bandwidth of GPU memory is 5–10 times as high as that of CPU memory, and a GPU thus excels in accelerating both compute- and memory-intensive scientific applications [31, 68, 84, 85]. However, as a GPU has a limited capacity of GPU memory (tens of GBs), it fails to directly run a large stencil code whose data size exceeds the memory capacity.

A large entity of research on GPU-based out-of-core stencil computation has been performed to address this issue [36, 37, 54, 82, 87, 90]. For a large dataset whose data size exceeds the capacity of the GPU memory, out-of-core computation first decomposes the dataset into smaller chunks and then streams the chunks to and from the GPU to process. Nevertheless, the performance of this approach is often limited by data transfer between the CPU and GPU because the interconnects fail to catch up with the development of the computation capability of GPUs as described in [85]. Data-centric strategies are thus necessary to reduce the data transfer.Studies have introduced strategies such as temporal blocking and region sharing to reuse the on-GPU data and to avoid extra data transfer [36, 37, 54, 82]. Nevertheless, according to [82], the performance of out-of-core code was still limited by data

transfer despite these strategies. Therefore, we need to further improve the methods to reduce data transfer time. A potential solution is to use on-the-fly compression to compress the data on the GPU before transferring it back to the CPU and decompress the data on the GPU before processing. However, hitherto studies on the acceleration of GPU-based out-of-core stencil computation with on-the-fly compression are really rare. According to a comprehensive review [10], studies on leveraging compression techniques in scientific applications mainly focused on scenarios such as post-analysis and failure recovery. We think that the scarcity of relevant research raises two research questions:

- Would the overhead of compression/decompression outweighs the reduced data transfer time?
- Would the precision loss involved by data compression be so huge that the output becomes useless?

In this study, we (1) propose a method to accelerate out-of-core stencil computation with on-the-fly compression on the GPU and (2) try to give answers to the two above-mentioned questions. The contributions of this work are as follows:

- We introduce a novel approach to integrate an on-the-fly lossy compression into the workflow of a five-point stencil computation. For large datasets that are decomposed into chunks, this approach solves the data dependency between contiguous chunks and thus secures the accessibility to the common regions between contiguous chunks after compression.
- We modify a widely-used GPU-based compression library [45] to support pipelining, which is mandatory for overlapping CPU-GPU data transfer with GPU computation.
- We analyze the experimental results to answer the aforementioned questions, i.e., on-the-fly compression is useful in reducing the overall execution time of out-of-core stencil computation, and the precision loss is tolerable.

The remainder of this study is organized as follows: Related studies on accelerating stencil and similar scientific applications with compression techniques are introduced in Section 4.2. Background of stencil computation and challenges in applying on-the-fly compression to stencil computation are briefly described in Section 4.3. Section 4.4 discusses the selection of an appropriate GPU-based compression library. The proposed method to integrate the compression processes into the workflow of out-of-core stencil computation is described in Section 4.5. In Section 4.6, experimental results are presented and analyzed. Finally, Section 4.7 concludes the present study and proposes future research directions.

4.2 Related Work

Nagayasu *et al.* [59] proposed a decompression pipeline to accelerate out-of-core volume rendering of time-varying data. Their method was specified to handle RGB data and the decompression procedure was partially performed on the CPU.

Tao *et al.* [94] proposed a lossy checkpointing scheme, which significantly improved the checkpointing performance of iterative methods with lossy compressors. Their scheme reduced the fault tolerance overhead for iterative methods by 23%–70% and 20%–58% compared to traditional checkpointing and lossless-compressed checkpointing, respectively.

Calhoun *et al.* [9] proposed metrics to evaluate the accuracy loss caused by using lossy compression to reduce the snapshot data used for checkpoint restart. They improved efficiency in checkpoint restart for partial differential equation (PDE) simulations by compressing the snapshot data and found that this compression did not affect overall accuracy in the simulation.

Wu *et al.* [104] proposed a method to simulate large quantum circuits using lossy or/and lossless compression techniques adaptively. They managed to increase the simulation size by 2–16 qubits. However, their method was designed for CPU-based supercomputers and thus the compression libraries cannot be used for GPU-based scenarios. Moreover, the adaptive selection between lossy and lossless compression, i.e., using lossy compression if lossless one failed, is impractical in GPU-based high-performance applications because such failures heavily impair the computational performance.

Jin *et al.* [38] proposed a method to use GPU-based lossy compression for extreme-scale cosmological simulations. Their findings show that GPU-based lossy compression can enable sufficient accuracy on post-analysis for cosmological simulations and high compression and decompression throughputs.

Tian *et al.* [95] proposed cuSZ, an efficient GPU-based error-bounded lossy compression framework for scientific computing. This framework reported high compression and decompression throughputs and a good compression ratio. However, according to their study, cuSZ has sequential subprocedures, which prevents us to use this framework as on-the-fly compression in our work due to the concern of the overhead to shift from GPU to CPU computation.

Zhou *et al.* [110] designed high-performance MPI libraries with on-the-fly compression for modern GPU clusters. In their work, they reduced the inter-node communication time by compressing the messages transferred between nodes, and the size of messages was up to 32 MB. On the other hand, our method compressed large datasets for stencil computation that were more than 10 GB to reduce the data transfer time between the CPU and GPU (i.e.,

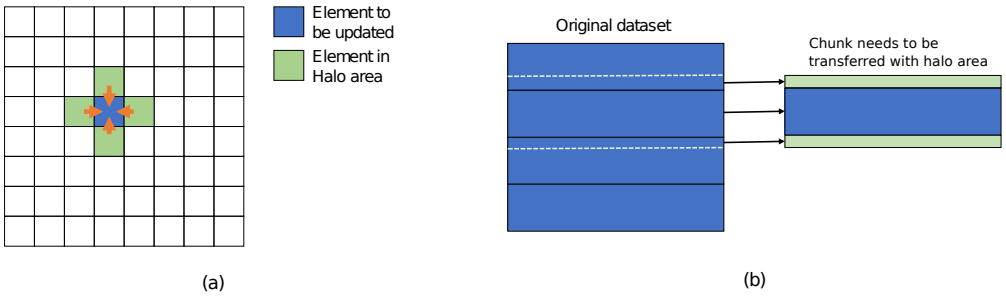

Figure 4.1: Five-point stencil computation: (a) updating each element based on the four neighboring elements, and (b) transferring each decomposed chunk with the halo data.

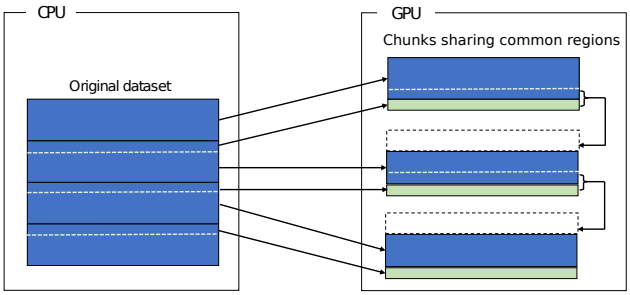

Figure 4.2: The contiguous chunks can share common regions on the GPU. By exploiting this characteristic, we can avoid transferring the amount of data equivalent to that of the halo areas.

intra-node communication time). Moreover, our method is specified to handle out-of-core stencil code, solving the data dependency between decomposed data chunks.

4.3 Out-of-Core Stencil Computation

Stencil computation is an iterative computation that updates each element of input datasets according to a fixed pattern that updates an element based on the elements surrounding it. A hello-world application of stencil computation is the solver of Laplace's equation, which can describe the phenomenon of heat conduction: A five-point stencil code, where the temperature of each data point at the $(t+1)$-th time step is obtained by taking the average temperature of the four surrounding points at the t-th time step (Figure 4.1(a)).

To use out-of-core approaches that handle excess data, we decompose the original datasets into smaller chunks and stream the chunks to and from the GPU for processing. Due to data dependency of stencil computation, when we transfer a chunk to the GPU, we must also piggyback the neighbor data ("halo area") with the chunk (Figure 4.1(b)). The size of halo data we must transfer along with the chunk increases in conformity with the number of time steps we want to process the chunk on the GPU. As two contiguous chunks share

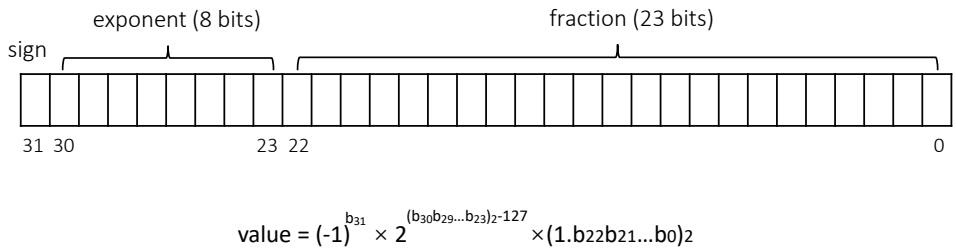

Figure 4.3: Single-precision floating-point format.

common regions, a chunk can get common regions from its former chunk as well as provide its later chunk with common regions. By doing so, we can effectively reduce the amount of data transfer equivalent to the size of halo data (Figure 4.2).

One challenge in integrating on-the-fly compression into the workflow of out-of-core stencil computation is that we must solve the aforementioned data dependency. Naively compressing each chunk not only consumes more memory space but also prevents sharing of common regions across contiguous chunks. Therefore, a sophisticated compression strategy is necessary, and will be introduced in Section 4.5.1.

4.4 On-the-Fly Compression

Another concern in leveraging on-the-fly compression in out-of-core stencil code is the overheads of compression and decompression that are often considerable. GPU-based compression libraries such as cuZFP [45], Cusz [95], and nvComp [66] report high speeds in compression and decompression. The cuZFP and Cusz libraries are based on lossy compression, whereas the nvComp is lossless.

In this study, we used cuZFP, a high-performance library with source code relatively easy to modify to implement functionalities we need. The library allows users to specify the number of bits used to preserve a value. For example, specifying 16 bits to preserve a single-precision floating-point (i.e., float-type) value achieves a compression ratio of 1/2. According to the format of float-type value shown in Figure 4.3 [39], we can also calculate the theoretical upper bound of point-wise relative error (PRE) for such a compression ratio. That is, the 0th to 15th bits are discarded. The PRE can be computed as

$$PRE = \frac{Discarded}{1 + Reserved + Discarded}, \tag{4.1}$$

where

$$Discarded = \sum_{i=0}^{15} b_i \times 2^{i-23}, \tag{4.2}$$

and

$$Preserved \;=\; \sum_{i=16}^{22} b_i \times 2^{i-23}. \tag{4.3}$$

To yield the upper bound, the 0th to 15th bits should be set ones to make up the greatest amount of discarded information, and the 16th to 22nd bits should be set zeros to make up the smallest amount of preserved information. In doing so, we obtain 0.008 as the upper-bound PRE. Nevertheless, such a theoretical upper bound will not occur in practice, because sequences of zeros and ones are in fact easily to be compressed.

We avoided using the lossless nvComp due to the concern of compression ratio. In our preliminary experiments, we found the size of data compressed with nvComp was larger than that of the original data. Therefore, we chose not to use nvComp in the present study because we could not estimate the upper bound of the size of the compressed data, and we must allocate GPU memory every time the compression happens instead of reusing pre-allocated GPU buffers with fixed sizes. The reason why we avoided using Cusz was explained in Section 4.2.

4.5 Proposed Method

In this section, we introduce our proposed method, including separate compression that solves the data dependency between contiguous chunks and thus allows us to compress the decomposed datasets freely, and a pipelining version of cuZFP that supports overlapping compression/decompression with CPU-GPU data transfer.

4.5.1 Separate Compression

As shown in Figure 4.2, two contiguous chunks have common regions that are shareable. The bottom halo areas needed by the i-th chunk lie in the $(i+1)$-th chunk, and the top halo areas needed by the $(i+1)$-th chunk lie in the i-th chunk. Therefore, the common region between the two chunks consists of the top areas and a part of the $(i+1)$-th chunk whose size is equivalent to that of the top halo areas. If we transfer the i-th chunk together with its bottom halo areas, we can avoid transferring the common regions for the $(i+1)$-th chunk.

Similarly, each chunk only needs to be transferred with its remainder and bottom halo areas, so the two parts, i.e., the remainder and half of the common region, must be exclusively readable and writable to the according contiguous chunks. Based on this observation, we propose a separate compression approach that compresses the two parts separately. As shown in Figure 4.4(a), prior to computation, the i-th compressed remainder and the common region are decompressed, therefore the i-th chunk can be computed on and provides the data needed

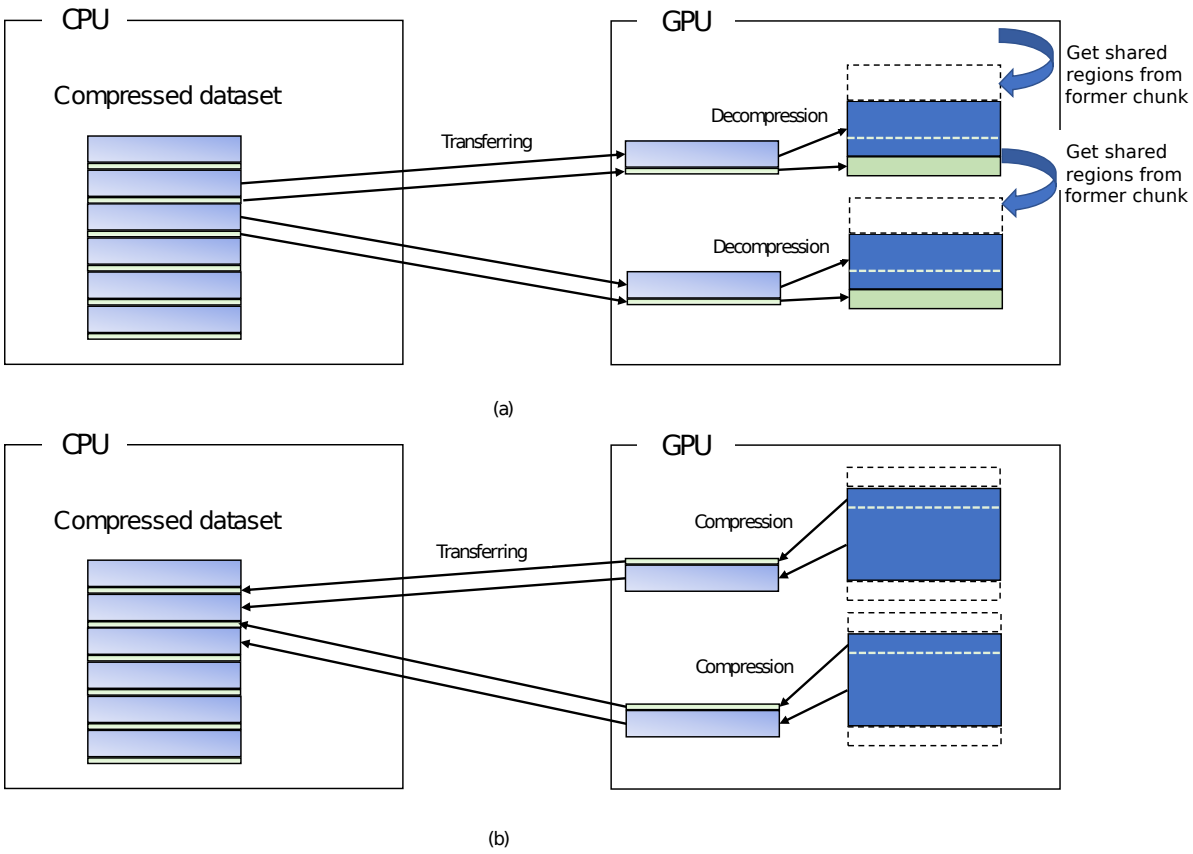

Figure 4.4: Separate compression approach to solve data dependency between contiguous chunks. In this approach, the remainder and the common region are compressed separately for each chunk. As shown in (a), the i-th compressed remainder and common region are decompressed on the GPU for computation; and in (b), after computation, the remainder and common region are compressed and transferred back to CPU to update the i-th remainder and $(i-1)$-th common region, respectively.

by the $(i+1)$-th chunk. As shown in Figure 4.2(b), after computation, the $(i+1)$-th chunk is compressed as the $(i+1)$-th remainder and i-th common region.

4.5.2 Pipelining cuZFP

The cuZFP library [45] is mainly designed as a standalone tool that can be seamlessly used for post-analysis and CPU-centric scientific computations. However, as an on-the-fly process in the out-of-core stencil computation, we have to modify the source code to support pipelining that overlaps CPU-GPU data transfer with GPU computations. Thanks to the good maintenance of the cuZFP project, we managed to modify the source code to add such functionality with a reasonable amount of programming effort. In pipelining cuZFP, we use three CUDA [65] streams (Figure 4.5).

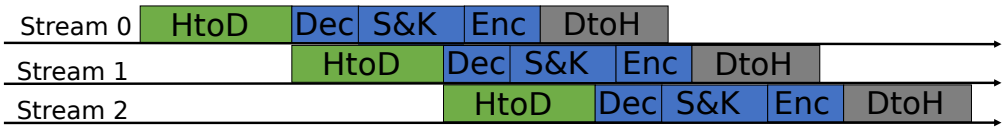

Figure 4.5: Modified cuZFP that supports pipelining. Three CUDA streams are used to perform operations, overlapping CPU-GPU data transfer (i.e., "HtoD" and "DtoH") with GPU kernels including compression (i.e., encoding, "Enc"), decompression (i.e., decoding, "Dec"), and computation (i.e., region sharing and computation kernel, "S&K").

Table 4.1: Testbed for experiments.

GPU	NVIDIA Tesla V100-PCIe
GPU memory	16 GB
CPU	Xeon Silver 4114
CPU memory	500 GB
OS	CentOS 7.1.0
CUDA	10.1
cuZFP [45]	0.5.5

4.6 Experimental Results

In this section, we analyze the experimental results to evaluate the benefits of using on-the-fly compression in out-of-core stencil computation on a GPU. The stencil code we used is an example code in the Formura stencil framework [58] which is a 2D five-point stencil computation for fluid simulation. In the simulation, a velocity field and a pressure field are updated relying on each other for each time step. In the experiments, we set the size of the field to 45,000×45,000 which sums up to 30 GB. Moreover, we used three codes in our experiments to evaluate the performance and fidelity:

1. A CPU-based code using OpenMP [96] multi-threading (40 threads).
2. A GPU-based code without compression.
3. A GPU-based code with the pressure field compressed using a 16/32 rate (i.e., using 16 bits to preserve each float value).

A feasible setting was used to execute the stencil codes, i.e., the number of divisions is 8 and the number of temporal blocking time steps is 16. Accordingly, we divide the data into 8 chunks, and when a chunk is transferred to the GPU, it will be computed 16 times before being transferred back to the CPU. For specifications of the testbed for all experiments performed, see Table 2.4.

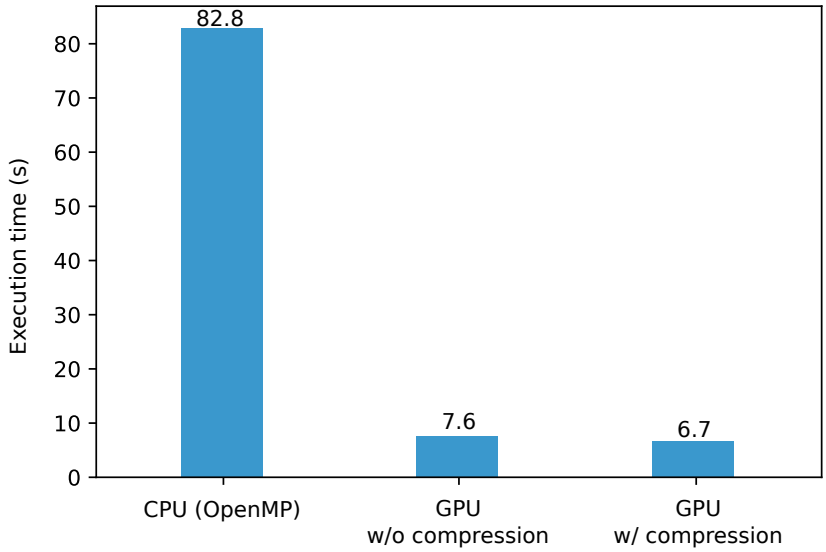

Figure 4.6: Execution time of the three stencil codes to run 64 time steps.

4.6.1 Evaluation of Performance Benefit

In this experiment, we ran the three codes for 64 time steps. As shown in Figure 4.6, the GPU code using on-the-fly compression achieved a speedup of 12.4×, compared to the CPU multi-threading code. Moreover, the GPU code with compression achieved a speedup of 1.13×, compared to the GPU code without compression, demonstrating our proposed method is beneficial for out-of-core GPU stencil computation in terms of performance.

4.6.2 Evaluation of Fidelity Loss

In this experiment, we ran the two GPU codes up to 384 time steps. We compared the output of the code using compression with that of the code without compression to evaluate the fidelity of the proposed method. In detail, for every 64 time steps, we sampled 20 million points of the pressure field (450 points for each row of the field) and compared the values of the GPU code using compression with that of the GPU code without compression to obtain the max and the average PREs. As shown in Figure 4.7, the max and the average PREs were relatively large at the beginning of the computation, but they decreased over time and finally became small and stable as the values of the stencil computation converged. Observing the results, we conclude the fidelity loss involved by the proposed method is tolerable.

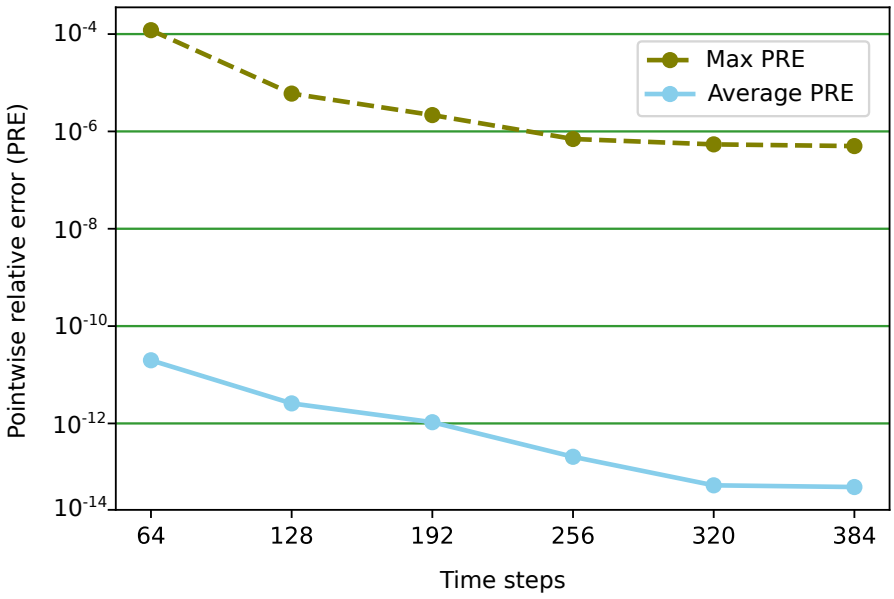

Figure 4.7: Change in fidelity loss from 64 to 384 time steps.

4.7 Conclusion

In this study, we introduced a method to accelerate out-of-core GPU stencil computation with on-the-fly compression. To realize the method, we proposed a novel approach to compress the decomposed data, solving the data dependency between contiguous chunks. We also modified the cuZFP library [45] to support pipelining for overlapping data transfer with GPU computation. Experimental results show that the proposed method achieved a speedup of 1.13×, compared to the method without compression. The fidelity loss caused by compression is tolerable, i.e., an average PRE smaller than 10^{-12} when the values of the stencil computation converge.

The results give preliminary answers to the two research questions mentioned in Section 4.1. First, the reduction of CPU-GPU data transfer time achieved by using on-the-fly compression outweighs the overhead of compression/decompression, improving the overall performance of out-of-core GPU stencil computation. Secondly, under the experimental settings in this work, on-the-fly compression does not cause severe accuracy-related problems up to 384 time steps.

Future work includes comparing other on-the-fly compression algorithms to cuZFP and creating a performance model to evaluate the benefits of applying on-the-fly compression to various stencil codes. Moreover, although the 2D fluid simulation code converges successfully in this work, there may be stencil codes that are more sensitive to accuracy loss. Therefore,

we plan to explore more stencil codes to generalize the use of on-the-fly compression by both conducting experiments and analyzing the characteristics of the corresponding PDEs.

Chapter 5

Conclusions

In this chapter, we summarize our work and discuss future work.

5.1 Summary of This Thesis

In this thesis, we investigated two out-of-core GPU applications: B&B 0-1 knapsack solver (Chapter 2) and stencil computation (Chapter 3 and 4). Both works focus on enabling those algorithms to process large-scale data on GPUs. Although the two applications have different characteristics such as static or variable memory footprint at runtime, we proposed techniques following the same mindset of data-centric computing, focusing on reusing the on-GPU data and reducing the CPU-GPU data transfer.

In Chapter 2, a GPU B&B solver for large 0-1 knapsack problems was presented. The solver streams subproblems to and from the GPU to relax the limitation of GPU memory capacity, and thus suffers from a large amount of CPU-GPU data transfer. To reduce the data transfer that limits the overall performance of the solver, we proposed two data-centric computing techniques including: (1) an O3S technique that minimizes the number of subproblems to be transferred per iteration, (2) a GPU stream compaction technique that condenses sparse datasets, and (3) an explicitly-managed pipelining technique that hides data transfer overhead by overlapping data transfer with GPU computation. Experimental results demonstrate that the out-of-core GPU solver stored 41× as many subproblems at a time, solving approximately twice as many instances of the 0-1 knapsack problem as a previous in-core solver. Moreover, the O3S technique helped the solver to solve more large instances by suppressing subproblem splitting. The O3S technique ran 12.2× as fast as the previous BFS strategy, contributing to a 7.5× speedup in terms of the entire execution time. Furthermore, the GPU stream compaction technique ran 9.9× as fast as a previous CPU stream compaction strategy.

In Chapter 3, we extended the PACC framework with two data-centric computing tech-

Table 5.1: Candidate combinatorial-optimization applications to use proposed out-of-core data-centric techniques.

Name	Potential baseline	Challenges
Generalized minimum spanning tree	[26]	Edges and vertices (equivalent to items in 0-1 KP) are pruned besides subproblems, and thus using atomic operations or modifying the original algorithm may be necessary
Maximum edge-weight clique problem	[86]	Recursive function-calls should be replaced with explicit subproblem-management
Flow-shop scheduling problem	[12]	The branching operation was performed on the CPU; therefore GPU-based branching should be implemented for higher overall performance

niques: (1) a direct-mapping technique to eliminate data copy between the original data and CPU buffers and (2) a region-sharing technique to avoid transferring halo regions between the CPU and GPU. Experimental results demonstrate that out-of-core stencil code generated by the extended framework outperformed codes based on OpenMP and Unified Memory by a factor of ten. Comparing the code using proposed techniques to the code without the techniques, we found the direct-mapping technique contributed to a 2.7× speedup and the region-sharing technique contributed to an additional 1.4× speedup.

With respect to the extent of programming effort reduction, we determined that the PACC framework automatically extended the original serial code 2.3 times in length to obtain the out-of-core parallel code. In addition, 75% of the extended code was different from the original serial code.

In Chapter 4, we proposed an additional data-centric computing technique to complement Chapter 3, further accelerating the out-of-core GPU stencil computation with on-the-fly GPU compression. To implement the technique, we designed a novel approach to compress the decomposed data and modified a widely used GPU compression library to support pipelining. Experimental results show that the stencil code with the proposed technique achieved a speedup of 1.13× with a tolerable fidelity loss, compared to the stencil code without compression.

5.2 Future Work

As mentioned in Section 1.3, although this thesis focuses on reducing the data transfer between the CPU and GPU, the proposed data-centric computing techniques can also be applied to applications running in a multi-node environment for reducing the inter-node data transfer. Extending the proposed techniques to a multi-node environment allows us to handle larger data than using the techniques with a single pair of CPU and GPU.

Generalizing the proposed data-centric techniques is also a promising direction for fu-

ture research. In addition to the 0-1 knapsack problem, we plan to target other large-scale combinatorial-optimization problems such as traveling salesman problem and minimum spanning tree. Table 5.1 gives three candidate combinatorial-optimization applications to test the proposed out-of-core data-centric techniques. Moreover, in addition to stencil computation, we plan to target other scientific applications such as quantum computing simulation and sequence alignment.

Last but not least, although slow compared to GPU computing capability, the interconnect bandwidth also improves over time. We thus need to build a performance model to evaluate how much a large GPU application can benefit from data-centric computing techniques, considering the characteristics of the application, the GPU computing capability, and the interconnect bandwidth. Such a model is important not only for determining the configuration to run the application but also for designing novel data-centric techniques.

Bibliography

[1] S. Adams, J. Payne, and R. Boppana. Finite difference time domain (FDTD) simulations using graphics processors. In *Proceedings of the High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC)*, pages 334–338, 2007.

[2] V. Allada, T. Benjegerdes, and B. Bode. Performance analysis of memory transfers and GEMM subroutines on NVIDIA Tesla GPU cluster. In *Proceedings of the 11th IEEE International Conference on Cluster Computing and Workshops (CLUSTERW)*, pages 1–9, 2009.

[3] R. Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[4] M. Berrada and K. E. Stecke. A branch and bound approach for machine load balancing in flexible manufacturing systems. *Management Science*, 32(10):1316–1335, 1986.

[5] C. Bonati, S. Coscetti, M. D ' Elia, M. Mesiti, F. Negro, E. Calore, S. F. Schifano, G. Silvi, and R. Tripiccione. Design and optimization of a portable LQCD Monte Carlo code using OpenACC. *International Journal of Modern Physics C*, 28(05):1750063, 2017.

[6] U. Bondhugula, V. Bandishti, and I. Pananilath. Diamond tiling: tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1285–1298, 2016.

[7] A. Boukedjar, M. E. Lalami, and D. El-Baz. Parallel branch and bound on a CPU-GPU system. In *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 392–398, 2012.

[8] H. Cai, L. Zhu, and S. Han. ProxylessNAS: direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[9] J. Calhoun, F. Cappello, L. N. Olson, M. Snir, and W. D. Gropp. Exploring the feasibility of lossy compression for PDE simulations. *The International Journal of High Performance Computing Applications*, 33(2):397–410, 2019.

[10] F. Cappello, S. Di, and A. M. Gok. Fulfilling the promises of lossy compression for scientific applications. In *Proceedings of Smoky Mountains Computational Sciences and Engineering Conference (SMC)*, pages 99–116, 2020.

[11] T. Carneiro, A. E. Muritiba, M. Negreiros, and G. A. L. de Campos. A new parallel schema for branch-and-bound algorithms using GPGPU. In *Proceedings of the 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 41–47, 2011.

[12] I. Chakroun and N. Melab. An adaptative multi-GPU based branch-and-bound. a case study: the flow-shop scheduling problem. In *Proceedings of the IEEE 14th International Conference on High Performance Computing and Communication & IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, pages 389–395, 2012.

[13] Y. Chen, F. Hua, Y. Jin, and E. Z. Zhang. BGPQ: a heap-based priority queue design for GPUs. In *Proceedings of the 50th International Conference on Parallel Processing (ICPP)*, pages 1–10, 2021.

[14] G. B. Dantzig. Discrete-vriable extremum problems. *Operations Research*, 5(2):266–288, 1957.

[15] K. Datta. *Auto-tuning stencil codes for cache-based multicore platforms*. University of California, Berkeley, 2009.

[16] S. Deldon, J. Beyer, and D. Miles. OpenACC and CUDA unified memory. *Proceedings of Cray User Group (CUG)*, 2018.

[17] J. Eckstein, C. A. Phillips, and W. E. Hart. PICO: an object-oriented framework for parallel branch and bound. In *Studies in Computational Mathematics*, volume 8, pages 219–265. Elsevier, 2001.

[18] T. Endo. Applying recursive temporal blocking for stencil computations to deeper memory hierarchy. In *Proceedings of the IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 19–24, 2018.

[19] A. E. Ezugwu, V. Pillay, D. Hirasen, K. Sivanarain, and M. Govender. A comparative study of meta-heuristic optimization algorithms for 0–1 knapsack problem: some initial results. *IEEE Access*, 7:43979–44001, 2019.

[20] A. Farrés, C. Rosas, M. Hanzich, M. Jordà, and A. Peña. Performance evaluation of fully anisotropic elastic wave propagation on NVIDIA Volta GPUs. In *Proceedings of the 81st EAGE Conference and Exhibition (EAGE)*, volume 2019, pages 1–5, 2019.

[21] B. Fornberg. Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of Computation*, 51(184):699–706, 1988.

[22] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, and G. Yang. The Sunway TaihuLight supercomputer: system and applications. *Science China Information Sciences*, 59(7):1–16, 2016.

[23] J.-P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth. Master-worker: an enabling framework for applications on the computational grid. *Cluster Computing*, 4(1):63–70, 2001.

[24] W. Gropp, W. D. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999.

[25] T. Grosser, A. Cohen, P. H. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. Split tiling for GPUs: automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU)*, pages 24–31, 2013.

[26] M. Haouari, J. Chaouachi, and M. Dror. Solving the generalized minimum spanning tree problem by a branch-and-bound algorithm. *Journal of the Operational Research Society*, 56(4):382–389, 2005.

[27] M. Harris. CUDA 9 features revealed: Volta, cooperative groups and more. https://developer.nvidia.com/blog/cuda-9-features-revealed/, 2017.

[28] S. Heldens, P. Hijma, B. V. Werkhoven, J. Maassen, A. S. Belloum, and R. V. Van Nieuwpoort. The landscape of exascale research: a data-driven literature analysis. *ACM Computing Surveys*, 53(2):1–43, 2020.

[29] K. Hou, H. Wang, and W. Feng. GPU-unicache: Automatic code generation of spatial blocking for stencils on GPUs. In *Proceedings of ACM International Conference on Computing Frontiers (CF)*, pages 107–116, 2017.

[30] M. Hristakeva and D. Shrestha. Solving the 0-1 knapsack problem with genetic algorithms. In *Proceedings of the Midwest Instruction and Computing Symposium (MICS)*, pages 16–17, 2004.

[31] K. Ikeda, F. Ino, and K. Hagihara. Efficient acceleration of mutual information computation for nonrigid registration using CUDA. *IEEE Journal of Biomedical and Health Informatics*, 18(3):956–968, 2014.

[32] K. Ikeda, F. Ino, and K. Hagihara. An OpenACC optimizer for accelerating histogram computation on a GPU. In *Proceedings of the 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 468–477, 2016.

[33] F. Ino, Y. Munekawa, and K. Hagihara. Sequence homology search using fine grained cycle sharing of idle GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 23(4):751–759, 2011.

[34] F. Ino, K. Shigeoka, T. Okuyama, M. Motokubota, and K. Hagihara. A parallel scheme for accelerating parameter sweep applications on a GPU. *Concurrency and Computation: Practice and Experience*, 26(2):516–531, 2014.

[35] Jared Hoberock and Nathan Bell. Thrust: a productivity-oriented library for CUDA. http://thrust.github.io/, 2011.

[36] G. Jin, T. Endo, and S. Matsuoka. A multi-level optimization method for stencil computation on the domain that is bigger than memory capacity of GPU. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium, Workshops, and PhD Forum (IPDPSW)*, pages 1080–1087, 2013.

[37] G. Jin, J. Lin, and T. Endo. Efficient utilization of memory hierarchy to enable the computation on bigger domains for stencil computation in CPU-GPU based systems. In *Proceedings of the International Conference on High Performance Computing and Applications (ICHPCA)*, pages 1–6, 2014.

[38] S. Jin, P. Grosset, C. M. Biwer, J. Pulido, J. Tian, D. Tao, and J. Ahrens. Understanding GPU-based lossy compression for extreme-scale cosmological simulations. In *Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 105–115, 2020.

[39] W. Kahan. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.

[40] J. A. Kahle, J. Moreno, and D. Dreps. 2.1 Summit and Sierra: designing AI/HPC supercomputers. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, pages 42–43, 2019.

[41] Khronos OpenCL Working Group. The OpenCL specification. https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf, 2019.

[42] M. E. Lalami and D. El-Baz. GPU implementation of the branch and bound method for knapsack problems. In *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium, Workshops, and PhD Forum (IPDPSW)*, pages 1769–1777, 2012.

[43] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. In *50 Years of Integer Programming 1958–2008*, pages 105–132. Springer, 2010.

[44] J. Lin and J. A. Storer. Processor-efficient hypercube algorithms for the knapsack problem. *Journal of Parallel and Distributed Computing*, 13(3):332–337, 1991.

[45] P. Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, 2014.

[46] Y. Lu, F. Ino, and K. Hagihara. Cache-aware GPU optimization for out-of-core cone beam CT reconstruction of high-resolution volumes. *IEICE Transactions on Information and Systems*, 99(12):3060–3071, 2016.

[47] D. Luebke and G. Humphreys. How GPUs work. *Computer*, 40(2):96–100, 2007.

[48] Mark Harris. How to optimize data transfers in CUDA C/C++. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/, 2012.

[49] S. Martello. Knapsack problems: algorithms and computer implementations. *Wiley-Interscience Series in Discrete Mathematics and Optimization*, 1990.

[50] S. Martello, D. Pisinger, and P. Toth. *Dynamic programming and tight bounds for the 0-1 knapsack problem*. Datalogisk Institut, Københavns Universitet, 1997.

[51] S. Martello, D. Pisinger, and P. Toth. New trends in exact algorithms for the 0–1 knapsack problem. *European Journal of Operational Research*, 123(2):325–332, 2000.

[52] J. McCalpin and D. Wonnacott. Time skewing: a value-based approach to optimizing for memory locality. *Technical Report DCS-TR-379*, 1998.

[53] N. Miki, F. Ino, and K. Hagihara. An extension of OpenACC directives for out-of-core stencil computation with temporal blocking. In *Proceedings of the 3rd Workshop on Accelerator Programming Using Directives (WACCPD)*, pages 36–45, 2016.

[54] N. Miki, F. Ino, and K. Hagihara. PACC: a directive-based programming framework for out-of-core stencil computation on accelerators. *International Journal of High Performance Computing and Networking*, 13(1):19–34, 2019.

[55] Y. Mitani, F. Ino, and K. Hagihara. Parallelizing exact and approximate string matching via inclusive scan on a GPU. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):1989–2002, 2016.

[56] G. R. Mudalige, I. Reguly, S. P. Jammy, C. T. Jacobs, M. B. Giles, and N. D. Sandham. Large-scale performance of a DSL-based multi-block structured-mesh application for direct numerical simulation. *Journal of Parallel and Distributed Computing*, 131:130–146, 2019.

[57] T. Muranushi and J. Makino. Optimal temporal blocking for stencil computation. *Procedia Computer Science*, 51:1303–1312, 2015.

[58] T. Muranushi, S. Nishizawa, H. Tomita, K. Nitadori, M. Iwasawa, Y. Maruyama, H. Yashiro, Y. Nakamura, H. Hotta, J. Makino, et al. Automatic generation of efficient codes from mathematical descriptions of stencil computation. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing (FHPC)*, pages 17–22, 2016.

[59] D. Nagayasu, F. Ino, and K. Hagihara. A decompression pipeline for accelerating out-of-core volume rendering of time-varying data. *Computers & Graphics*, 32(3):350–362, 2008.

[60] Nikolay Sakharnykh. Maximizing unified memory performance in CUDA. https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/, 2017.

[61] NVIDIA Corporation. NVIDIA Nsight Visual Studio edition 4.7 user guide. https://docs.nvidia.com/nsight-visual-studio-edition/4.7/Nsight_Visual_Studio_Edition_User_Guide.htm, 2015.

[62] NVIDIA Corporation. NVIDIA Tesla V100 GPU architecture. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[63] NVIDIA Corporation. CUDA toolkit documentation v9.0.176. https://docs.nvidia.com/cuda/archive/9.0/, 2018.

[64] NVIDIA Corporation. CUDA C programming guide v10.1. https://docs.nvidia.com/cuda/archive/10.1/pdf/CUDA_C_Programming_Guide.pdf, 2019.

[65] NVIDIA Corporation. CUDA C++ programming guide 11.5.0. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2021.

[66] NVIDIA Developer. nvComp: high speed data compression using NVIDIA GPUs. https://developer.nvidia.com/nvcomp, 2019.

[67] NVIDIA Research. CUB documentation. https://nvlabs.github.io/cub/.

[68] T. Okuyama, M. Okita, T. Abe, Y. Asai, H. Kitano, T. Nomura, and K. Hagihara. Accelerating ODE-based simulation of general and heterogeneous biophysical models using a GPU. *IEEE Transactions on Parallel and Distributed Systems*, 25(8):1966–1975, 2013.

[69] M. Pedemonte, E. Alba, and F. Luna. Towards the design of systolic genetic search. In *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium, Workshops, and PhD Forum (IPDPSW)*, pages 1778–1786, 2012.

[70] PGI Compilers & Tools. OpenACC getting started guide. https://www.pgroup.com/resources/docs/19.1/pdf/openacc19_gs.pdf, 2019.

[71] T. Pupko, I. Pe'er, M. Hasegawa, D. Graur, and N. Friedman. A branch-and-bound algorithm for the inference of ancestral amino-acid sequences when the replacement rate varies among sites: Application to the evolution of five gene families. *Bioinformatics*, 18(8):1116–1123, 2002.

[72] Z. Quan and L. Wu. Design and evaluation of a parallel neighbor algorithm for the disjunctively constrained knapsack problem. *Concurrency and Computation: Practice and Experience*, 29(20):e3848, 2017.

[73] T. K. Ralphs. Parallel branch and cut for capacitated vehicle routing. *Parallel Computing*, 29(5):607–629, 2003.

[74] I. Z. Reguly, G. R. Mudalige, and M. B. Giles. Beyond 16GB: out-of-core stencil computations. In *Proceedings of the Workshop on Memory Centric Programming for HPC (MCHPC)*, pages 20–29, 2017.

[75] I. Z. Reguly, G. R. Mudalige, and M. B. Giles. Loop tiling in large-scale stencil codes at run-time with OPS. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):873–886, 2017.

[76] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith. The OPS domain specific abstraction for multi-block structured grid computations. In *Proceedings of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 58–67, 2014.

[77] Ryutaro Himeno. Himeno benchmark. https://i.riken.jp/en/ supercom/documents/himenobmt/, 2015.

[78] A. Schäfer and D. Fey. High performance stencil code algorithms for GPGPUs. *Procedia Computer Science*, 4:2027–2036, 2011.

[79] D. Schor. ORNL ' s 200-petaFLOPS summit supercomputer has arrived, to become world ' s fastest. https://fuse.wikichip.org/news/1351/ornls-200-petaflops-summit-supercomputer-has-arrived-to-become-worlds-fastest, 2018.

[80] M. S. Serpa, E. H. Cruz, M. Diener, A. M. Krause, A. Farrés, C. Rosas, J. Panetta, M. Hanzich, and P. O. Navaux. Strategies to improve the performance of a geophysics model for different manycore systems. In *Proceedings of the International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 49–54, 2017.

[81] J. M. Shalf and R. Leland. Computing beyond Moore's Law. *Computer*, 48(12):14–23, 2015.

[82] J. Shen, F. Ino, A. Farrés, and M. Hanzich. A data-centric directive-based framework to accelerate out-of-core stencil computation on a GPU. *IEICE Transactions on Information and Systems*, 103(12):2421–2434, 2020.

[83] J. Shen, J. Mei, M. Walldén, and F. Ino. Integrating GPU support for FreeSurfer with OpenACC. In *Proceedings of the IEEE 6th International Conference on Computer and Communications (ICCC)*, pages 1622–1628, 2020.

[84] J. Shen, K. Shigeoka, F. Ino, and K. Hagihara. An out-of-core branch and bound method for solving the 0-1 knapsack problem on a GPU. In *Proceedings of the 17th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 254–267, 2017.

[85] J. Shen, K. Shigeoka, F. Ino, and K. Hagihara. GPU-based branch-and-bound method to solve large 0-1 knapsack problems with data-centric strategies. *Concurrency and Computation: Practice and Experience*, 31(4):e4954, 2019.

[86] S. Shimizu, K. Yamaguchi, and S. Masuda. A branch-and-bound based exact algorithm for the maximum edge-weight clique problem. In *Proceedings of the 5th International Conference on Computational Science/Intelligence & Applied Informatics (CCSI)*, pages 27–47, 2018.

[87] T. Shimokawabe, T. Endo, N. Onodera, and T. Aoki. A stencil framework to realize large-scale computations beyond device memory capacity on GPU supercomputers. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, pages 525–529, 2017.

[88] P. Siegl, R. Buchty, and M. Berekovic. Data-centric computing frontiers: A survey on processing-in-memory. In *Proceedings of the 2nd International Symposium on Memory Systems (MEMSYS)*, pages 295–308, 2016.

[89] J. M. Silva, C. Boeres, L. M. Drummond, and A. A. Pessoa. Memory aware load balance strategy on a parallel branch-and-bound application. *Concurrency and Computation: Practice and Experience*, 27(5):1122–1144, 2015.

[90] M. Sourouri, S. B. Baden, and X. Cai. Panda: a compiler framework for concurrent CPU+GPU execution of 3d stencil computations on GPU-accelerated supercomputers. *International Journal of Parallel Programming*, 45(3):711–729, 2017.

[91] Steve Rennich. CUDA C/C++ streams and concurrency. https://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf, 2011.

[92] S. Tabik, M. Peemen, and L. F. Romero. A tuning approach for iterative multiple 3d stencil pipeline on GPUs: Anisotropic Nonlinear Diffusion algorithm as case study. *The Journal of Supercomputing*, 74(4):1580–1608, 2018.

[93] Y. Tanaka, M. Sato, M. Hirano, H. Nakada, and S. Sekiguchi. Performance evaluation of a firewall-compliant Globus-based wide-area cluster system. In *Proceedings of the 9th International Symposium on High-Performance Distributed Computing (HPDC)*, pages 121–128, 2000.

[94] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello. Improving performance of iterative methods by lossy checkponting. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 52–65, 2018.

[95] J. Tian, S. Di, K. Zhao, C. Rivera, M. Hickman Fulp, R. Underwood, S. Jin, X. Liang, J. Calhoun, D. Tao, and F. Cappello. cuSZ: an efficient GPU-based error-bounded lossy compression framework for scientific data. In *Proceedings of the 29th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2020.

[96] R. van der Pas, E. Stotzer, and C. Terboven. *Using OpenMP–the next step*. MIT Press, 2017.

[97] E. Vitali, D. Gadioli, G. Palermo, A. Beccari, C. Cavazzoni, and C. Silvano. Exploiting OpenMP and OpenACC to accelerate a geometric approach to molecular docking in heterogeneous HPC nodes. *The Journal of Supercomputing*, 75(7):3374–3396, 2019.

[98] L. Wan, K. Li, J. Liu, and K. Li. Efficient CPU-GPU cooperative computing for solving the subset-sum problem. *Concurrency and Computation: Practice and Experience*, 28(2):492–516, 2016.

[99] L. Wang, S. Chen, Y. Tang, and J. Su. Gregex: GPU based high speed regular expression matching engine. In *Proceedings of the 5th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, pages 366–370, 2011.

[100] Wikipedia contributors. Data-centric computing — Wikipedia, the free encyclopedia, 2021.

[101] Wikipedia contributors. History of supercomputing — Wikipedia, the free encyclopedia, 2021.

[102] Wikipedia Contributors. Supercomputer — Wikipedia, the free encyclopedia, 2021.

[103] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 171–180, 2000.

[104] X.-C. Wu, S. Di, E. M. Dasgupta, F. Cappello, H. Finkel, Y. Alexeev, and F. T. Chong. Full-state quantum circuit simulation by using data compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–24, 2019.

[105] W. Xue and C. J. Roy. Multi-GPU performance optimization of a computational fluid dynamics code using OpenACC. *Concurrency and Computation: Practice and Experience*, 33(5):e6036, 2021.

[106] I. Yamazaki, S. Tomov, and J. Dongarra. Non-GPU-resident symmetric indefinite factorization. *Concurrency and Computation: Practice and Experience*, 29(5):e4012, 2017.

[107] G. Zäpfel, R. Braune, and M. Bögl. The knapsack problem and straightforward optimization methods. In *Metaheuristic Search Concepts*, pages 7–29. Springer, 2010.

[108] J. C. Zavala-Díaz, M. A. Cruz-Chávez, J. López-Calderón, J. A. Hernández-Aguilar, and M. E. Luna-Ortíz. A multi-branch-and-bound binary parallel algorithm to solve the knapsack problem 0–1 in a multicore cluster. *Applied Sciences*, 9(24):5368, 2019.

[109] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. TOP-PIM: throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 85–98, 2014.

[110] Q. Zhou, C. Chu, N. Kumar, P. Kousha, S. Ghazimirsaeed, H. Subramoni, and D. Panda. Designing high-performance MPI libraries with on-the-fly compression for modern GPU clusters. In *Proceedings of the 35th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 444–453, 2021.