論文 / 著書情報
Article / Book Information

題目(和文)	現代的なGPUにおける計算性能とメモリバンド幅の乖離の克服
Title(English)	Overcoming the Gap Between Compute and Memory Bandwidth in Modern GPUs
著者(和文)	張 鈴啓
Author(English)	Lingqi Zhang
出典(和文)	学位:博士(理学), 学位授与機関:東京工業大学, 報告番号:甲第12510号, 授与年月日:2023年9月22日, 学位の種別:課程博士, 審査員:遠藤 敏夫,増原 英彦,脇田 建,坂本 龍一,横田 理央,松岡 聡,Attia Mohamed Wahib Mohamed
Citation(English)	Degree:Doctor (Science), Conferring organization: Tokyo Institute of Technology, Report number:甲第12510号, Conferred date:2023/9/22, Degree Type:Course doctor, Examiner:,,,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Overcoming the Gap Between Compute and Memory Bandwidth in Modern GPUs

Lingqi Zhang

A DISSERTATION

Presented to the Tokyo Institute of Technology

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

in

Mathematics and Computing Science

2023

Supervisor of Dissertation

Toshio Endo

Co-Supervisor of Dissertation

Satoshi Matsuoka

ACKNOWLEDGEMENT

I want to thank my supervisors, Prof. Satoshi Matsuoka and Prof. Toshio Endo, for allowing me to study at Tokyo Institute of Technology and join Matsuoka Lab. This opportunity has allowed me to attend top-tier international conferences and domestic workshops and collaborate with brilliant minds from around the world. I would also like to thank Mohamed Wahib, who has been both a mentor and a friend, guiding me through numerous challenges. I want to express my gratitude to Jens Domke. I learned a lot of paper writing through our collaboration (secretly). I would also like to thank Chen Peng and Yosuke Oyama for being my role model.

I would also like to express my deep gratitude towards Keiko Yoshida-san and Akihiro Nomura-san, the research administrators at Matsuoka Lab, for their consistent assistance and support with administrative tasks and life in Japan. Furthermore, I would like to thank my friends and colleague in Matsuoka Lab for helping me with research and life in Japan.

I want to thank MEXT for providing me with a scholarship and the opportunity to come to Japan and study at Tokyo Institute of Technology. The Japanese course accompanying the scholarship help me better understand Japanese culture and enjoy life in Japan. I would also like to thank RIKEN Center for Computational Science, Tokyo Institute of Technology, and National Institute of Advanced Industrial Science and Technology for allowing me to work as an intern and research assistant. These positions are especially beneficial after the scholarship period comes to an end.

I want to thank my Master's Degree supervisor, Prof. Shoubin Dong. Without her support, it would not have been possible for me to take this leap of faith to study abroad. I also want to thank Mr. Zebang Chen for persuading me to come to Japan. Though it is impossible to assume what would have happened otherwise, I am sure that the last five years in Japan have been one of the most precious experiences in my life.

Lastly, I would also like to thank my parents, as I would not have come this far without their sincere support.

ABSTRACT

OVERCOMING THE GAP BETWEEN COMPUTE AND MEMORY BANDWIDTH IN MODERN GPUS

Lingqi Zhang

The imbalance between the computational speed of a processor and memory bandwidth was identified two decades ago. Plenty of architectural research has been undertaken to mitigate this issue. Nevertheless, the gap between compute and memory bandwidth continues to widen. As a result, many workloads are bound by memory instead of by compute. Such workloads are classified as memory-bound kernels, and numerous efforts have been expended to optimize these kernels.

This dissertation also centers on memory-bound kernels, with a particular emphasis on Graphics Processing Units (GPUs), given their rising prevalence in High-Performance Computing (HPC) systems. More than half of the systems in the Top500 include discrete GPUs, and seven out of the top ten systems are GPU-accelerated (November 2021 list).

In this dissertation, we initially focus on the evolution trend of GPU development in the last decades. Examples include cooperative groups (i.e., device-wide barriers), asynchronous copy of shared memory (i.e., hardware prefetching) low(er) latency of operations, and larger volume of on-chip resources (register files and L1 cache).

Based on the observations of the latest GPU developments, we present strategies for overcoming the imbalance in compute and memory bandwidth. Specifically, we propose to extend the lifetime of the kernel across the time steps and take advantage of the large volume of on-chip resources (i.e., register files and scratchpad memory) in *reducing* or *eliminating* traffic to the device memory. Furthermore, we champion a minimum level of parallelism to maximize the available on-chip resources, maximizing the impact of the first strategy.

Then, we examine the effect of these strategies through two different paths:

First, we propose a general execution model for running memory-bound iterative GPU kernels: PERsistent KernelS (PERKS). In this model, the time loop is moved inside a persistent kernel, and device-wide barriers are used for resolving dependency. We then reduce the traffic to device memory by caching a subset of the output in each time step in registers and shared memory to be used as input for the following time step. We demonstrate the effectiveness of PERKS for a wide range of iterative 2D/3D stencil benchmarks (geometric mean speedup of 2.29x in small domains and 1.53x in large domains), and a Krylov subspace solver (geometric mean speedup of 4.67x in smaller SpMV datasets from SuiteSparse and 1.39x in larger SpMV datasets, for conjugate gradient).

We also reexamine temporal blocking optimizations for GPUs, investigating how temporal blocking schemes can be adapted to incorporate the latest features of recent Nvidia GPUs. We propose a novel temporal blocking method, EBISU, which champions low device concurrency to drive aggressive deep temporal blocking on large tiles that are executed tile-by-tile. We compare EBISU with state-of-the-art temporal blocking libraries: STENCILGEN and AN5D. We also compare EBISU with state-of-the-art stencil auto-tuning tools equipped with temporal blocking optimizations: ARTEMIS and DRSTENCIL. Over a wide range of stencil benchmarks, EBISU achieves a geometric mean speedup of 2.0x over any state-of-the-art counterparts and a geometric mean speedup of 1.49x over the best state-of-the-art performance in each stencil benchmark.

While the methodologies and implementations in this research exhibit superior performance compared to alternatives, the strategies and principles introduced here apply to any memory-bound kernels. We view this study as a trailblazing initiative and hope it will inspire researchers to delve deeper into the strategies and principles introduced in this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

Listings

CHAPTER 1

Introduction

1.1. Motivation

Since 1995, a consistent trend has been reported indicating that the performance of processors has been increasing at a faster pace than memory access [23, 24] a trend that continues even today [25]. We extend the concept of machine balance [23] to represent the number of double precision operations required for each byte of memory access ($\frac{Flops}{Bytes}$) and list the machine balance of the mainstream processors in Table 1.1.

Table 1.1 shows that the current mainstream processors are significantly faster than memory access. Unfortunately, this gap is even wider with the emergence of matrix engines [26]. Such processors favor algorithms with higher operation intensity [27, 28], such as GEMM [26]. However, iterative stencils [5, 8, 29, 30] and implicit solvers [31–33], which represent a significant proportion of High Performance Computing (HPC) workloads [34], typically exhibit low arithmetic intensity [5]. Hence, these workloads are difficult to exploit the computation power entirely and are consequently classified as memory-bound kernels.

Numerous strategies have been proposed to address this issue. From a hardware perspective, high-performance memory bandwidth (HBM) was considered a potential solution [35] and is currently implemented in high-end GPUs such as V100 [36], A100 [20], and MI250X [22], even CPUs like A64FX [15]. However, HBM is still unable to match the speed of processors. Other efforts including Process In Memory [37–39] (also known as Near Data Processing [40]) and cache hierarchy [41–43]. Large last level cache [44] seems to be a promising future that is already applied to AMD CPU (AMD V-Cache [45]). From a software perspective, researchers are striving to push the bottleneck closer to Cache, Last Level Cache [46–49] or L1 Cache [6], or even to registers [1, 8]. Nevertheless, if the processor development trend remains unchanged, optimizing memory-bound kernels will remain a critical topic in the near future.

This dissertation also aims at memory-bound kernels, with a specific focus on GPUs for two primary reasons: 1) more than half the systems on the Top500 list [50] include discrete GPUs, and seven of the systems in the top ten are GPU-accelerated (November 2022 list); 2) mainstream GPUs generally exhibit more significant imbalance compared to mainstream CPUs (as Table 1.1 shows). Upon observing the evolution of GPUs and the features of the latest models, specifically, the increasing on-chip resources, ease of saturation, and support for more complicated logic, it has become apparent that there is a new potential for optimizing memory-bound kernels.

According to these observations, this dissertation applies two strategies designed to enhance operational intensity without touching the underlying algorithm: Firstly, we suggest extending the

Table 1.1: CPU and GPU specifications and machine balance

Platform	Launched Year	Memory (GB/s)	Compute (Tensor Core) (TFLOPS/s)	Balance(Tensor Core) $\frac{\text{Flops}}{\text{Bytes}}$
Fujistu (A64FX) [15]	2019	1024	3.4	3.32
Intel (Platinum 8368) [16,17]	2021	204.8	1.094	5.34
AMD (7773X) [18,19]	2022	204.8	2.253	11
Nvidia (A100) [20]	2020	1555	9.7 (19.5)	6.24 (12.54)
Nvidia (H100-SMX) [21]	2022	3000	30 (60)	10 (20)
AMD (MI250X) [22]	2021	3200	47.9 (95.7)	14.97 (29.91)

lifespan of the kernel across time steps and leveraging the large volume of register files and scratchpad memory to reduce or eliminate traffic to the device memory in between time steps. Secondly, we champion minimizing parallelism to free up more on-chip resources for the aforementioned purposes.

We apply these strategies and design a *general approach* and *revisit the temporal blocking optimizations*:

❶ We propose a *generic execution model* for running iterative solvers on GPUs to improve data locality by taking advantage of the large volume of register files and scratchpad memory in reducing traffic to the device memory. ***PERsistent KernelS (PERKS)*** have the time loop inside them, instead of the host, and use recently supported device-wide barriers (in CUDA) for resolving dependency. Next, we identify the cachable data in the solver: data that is the output of time step $k-1$ and input to time step k, as well as the repeatedly loaded constant data. Finally, we use either the scratchpad memory or registers (or both) to cache the data and reduce the traffic to the device memory. The basic concept and implementation of PERKS are relatively simple, which we argue is essential for encouraging scientists and engineers to adopt PERKS in their iterative solvers implemented for GPUs, and other architectures. That being said, a challenging aspect we address is a detailed analysis of how and why PERKS is practical. The analysis requires an understanding of the effect of concurrency on performance.

❷ We *revisit the temporal blocking optimizations*. Temporal Blocking is one of the well-studied optimizations for iterative stencils. Usually, the dependency along the time dimension is resolved by either: a) redundantly loading and computing cells from adjacent blocks [51–53], or b) using tiling methods of complex geometry (e.g., trapezoidal and hexagonal tiling) along the time dimension and restrict the parallelism due to the dependency between neighboring blocks [54–56]. Either way, temporal blocking puts pressure on on-chip resources that limit the depth of time steps temporal blocking can combine. We propose ***EBISU*** that instead of restricting resource usage to improve parallelism, we restrict parallelism to optimize the use of resources. EBISU supports low device occupancy to facilitate aggressive deep temporal blocking on large tiles, which are processed one tile at a time. With this principle, we can implement very deep temporal blocking, considered to be a challenge in GPU before 1. The long dependencies introduced by deep temporal blocking make latency hard to be hidden on throughput-optimized platforms like GPUs, and 2. The potential

introduction of register pressure [5].

1.2. Main Contributions

This dissertation makes several contributions:

- We propose a generic execution model PERKS. We provide principles to design and implement PERKS. We also use performance analysis to understand the extent of PERKS and address performance concerns. Then we use stencil and conjugate gradient kernels as a case study to show how they are ported to PERKS and evaluate their performance. In iterative 2D/3D stencil benchmarks PERKS achieves a geometric mean speedup of 2.12x for 2D stencils and 1.24x for 3D stencils over state-of-art libraries; In Krylov subspace conjugate gradient solver, PERKS achieve a geometric mean speedup of 4.86x in smaller SpMV datasets from SuiteSparse and 1.43x in larger SpMV datasets over a state-of-art library. For more details, please refer to **(Chapter 4)**.
- We propose EBISU, which champions low device occupancy to drive aggressive deep temporal blocking on large tiles that are executed tile-by-tile. We introduce the design principle and schemes that scale with resources. We also introduce the cost model that can enhance design decisions. Finally, we compare EBISU with state-of-the-art stencil auto-tuning tools equipped with temporal blocking optimizations: ARTEMIS and DRSTENCIL. Over a wide range of stencil benchmarks, EBISU achieves a geometric mean speedup of 2.0x over any state-of-the-art counterparts and a geometric mean speedup of 1.49x over the best state-of-the-art performance in each stencil benchmark. For more details, please refer to **(Chapter 5)**

1.3. Dissertation Outline

- **Introduction** This opening chapter sets the stage for our research by exploring machine balance's history and current state and its implications for memory-bound kernels. Subsequently, we outline our proposals and contributions.
- **Background** This chapter briefly introduces the theory of parallelism, GPU architectures, and the target memory-bound kernels addressed in this dissertation.
- **Microbenchmarking Nvidia GPUs: A Decade of Evolution Trends** In this chapter, we conduct an exhaustive investigation of the latest GPU platforms, utilizing microbenchmarks from existing research and our own specially designed microbenchmarks for synchronization instructions. Based on our observations, we propose strategies that combine time steps and maintain minimal parallelism to address machine imbalance.
- **PERKS: A Locality-Optimized Execution Model For Iterative Memory-Bound GPU Applications** This chapter presents the design principles of PERKS and conducts a performance analysis to understand its extent. We then demonstrate how we adapt a broad range of kernels, such as stencils and conjugate gradient solvers, to PERKS, showing its speedup through evaluations.

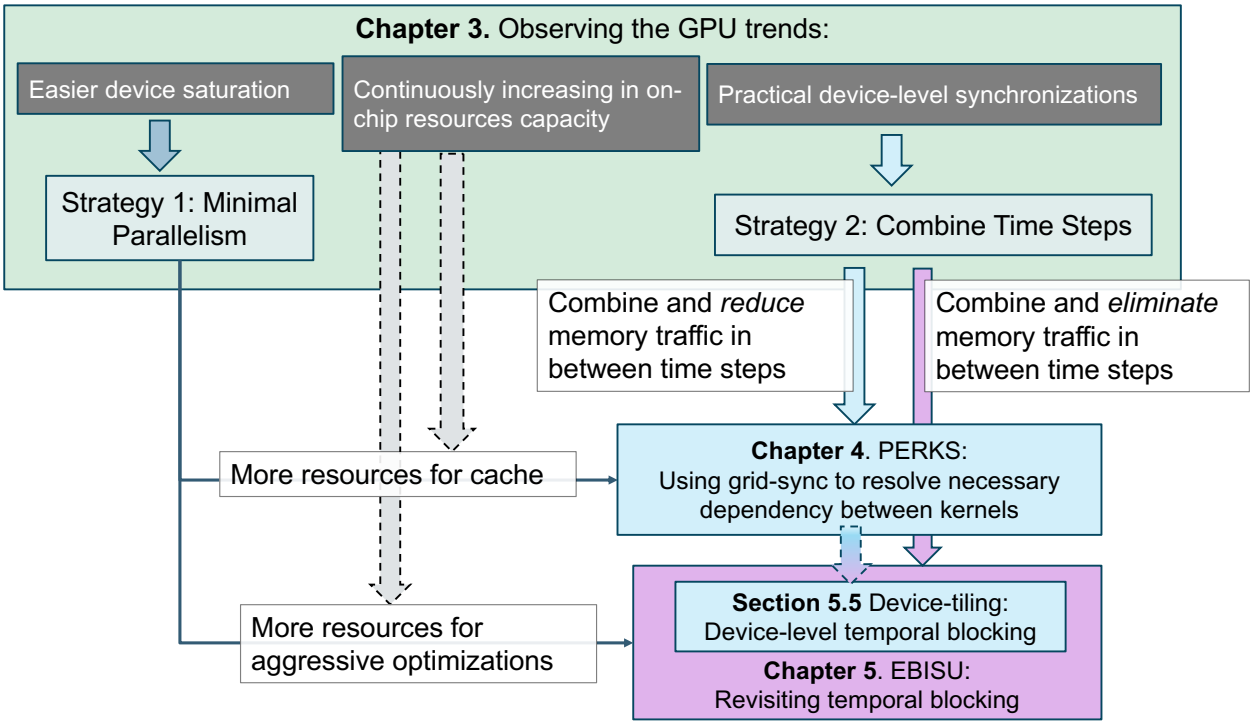


Figure 1.1: Overview of this dissertation.

- **EBISU: Epoch Blocking for Iterative Stencils, with Ultracompact Parallelism** In this chapter, we explain the design principle of EBISU and the analysis for enhancing the implementation decision. We showcase EBISU's performance by comparing it with a wide range of temporal blocking stencil implementations.
- **Discussion and Future Work** This chapter presents potential future topics that may stem from this research.
- **Conclusion** In this final chapter, we conclude this study and reflect on its findings

Figure 1.1 illuminate the overview of the relationship of the main contents in this dissertation.

CHAPTER 2

Background

2.1. Machine Balance & Roofline Model

The concepts of *Machine Balance* and the *Roofline Model* are fundamental to understanding hardware constraints and assessing achievable performance in this dissertation.

2.1.1. Machine Balance

Machine balance is a concept that dates back to 1988 [57]. John [23] use the representation $\frac{peak\ floating\ ops/sec}{sustained\ memory\ bandwidth\ ops/sec} = \frac{peak\ floating\ ops/sec}{peak\ memory\ ops/sec}$ for machine balance. We extend the concept of machine balance and use *flop per byte* to represent machine balance:

$$Balance = \frac{Peak\ flops\ GFLOPS/s}{Sustained\ memory\ bandwidth\ GB/s} \tag{2.1}$$

Machine balance measures the number of operations necessary per memory access operation.

2.1.2. Roofline Model

The roofline model is an insightful visual performance model that provides a simple and intuitive approach for predicting attainable performance [27, 58]. It introduced the concept of operational intensity I, which is inferred from Work W and the number of bytes of memory traffic incurred by executing a given program Q:

$$I = \frac{W}{Q} \tag{2.2}$$

The attainable performance $\mathbb{P}$ can be inferred from I and the hardware features:

$$\mathbb{P} = \min(Peak\ FLOPS, Peak\ Bandwidth \times I) \tag{2.3}$$

The roofline model is an intuitive tool for identifying potential bottlenecks. Some research further extends the roofline model to support cache [59] and tensor core [60].

2.1.3. Relationship Between Machine Balance and Roofline Model

The roofline model is a tool used to analyze the performance of a given kernel. Here, the operation intensity is a feature of the kernel. On the other hand, the machine balance is a feature of a given hardware. The roofline model and machine balance intersect when determining whether a given kernel is compute-bound or memory-bound:

$$A\ kernel\ is = \begin{cases} compute-bound, & \text{if } I \geq balance \\ memory-bound, & \text{otherwise} \end{cases} \tag{2.4}$$

Figure 2.1 gives an example of the roofline model. It draws the attainable bound for any kernels. We can observe a distinct turning point determined by the machine balance.

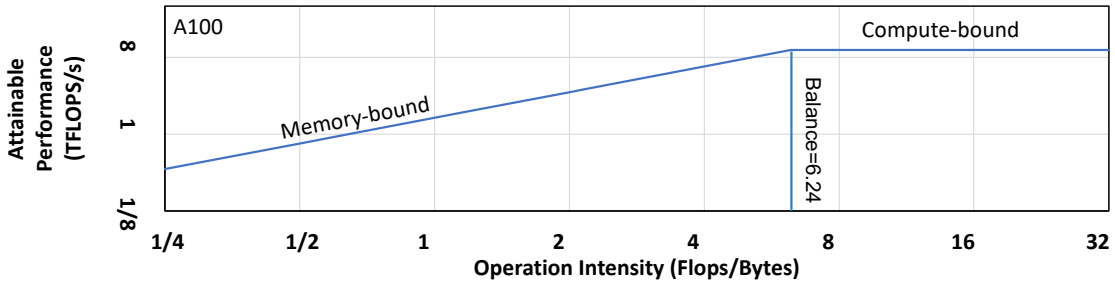


Figure 2.1: An example of roofline model for A100 GPU.

2.2. Little's Law

A. Cobham initially proposed the formulation of Little's Law in 1954 [61]. It delineates the relationship between the average number of items L in a system, the average arrival rate λ, and the average time W an item spends in the system as:

$$L = \lambda W \tag{2.5}$$

In the context of High-Performance Computing, we adopt an implication of Little's Law as presented by Gustafson [62]. This formulation uses latency $\mathbb{L}$ and throughput $\mathbb{THR}$ to infer the concurrency $\mathbb{C}$ of the given hardware:

$$\mathbb{C} = \mathbb{L} \times \mathbb{THR} \tag{2.6}$$

Here $\mathbb{L}$ is the latency of a single operator (corresponding to W); $\mathbb{THR}$ is the average operator processing rate (corresponding to λ). And $\mathbb{C}$ can be interpreted as the number of operator units that can be processed simultaneously (corresponding to L).

While Little's Law primarily provides insight into hardware features and is commonly used in computer design considerations [62], this dissertation uses it as a guideline for software design: we consider that software saturates the device if it provides a higher level of parallelism than the concurrency indicated by Little's Law.

2.3. Parallelisms

In this section, we define parallelism $\mathbb{PAR}$ as a feature of a kernel, distinct from the concept of concurrency $\mathbb{C}$ mentioned in Section 2.2. We consider concurrency as a feature of the hardware.

2.3.1. Thread Level Parallelism (TLP)

We adopt the concept of TLP used in paper [63], which refers to the number of activated threads per stream multiprocessor. It can be calculated as:

$$TLP = activeThreadBlock \times ThreadPerBlock \tag{2.7}$$

Kernel with high TLP usually exhibits superior performance [63], but it also implies fewer resources available per thread.

Listing 2.1: Pseudocode for 1D 3-Point Jacobian Stencil, ILP=1

```
1  for(int i=0; i<N; i++){
2      tmp=a*in[i-1];
3      tmp+=b*in[i];
4      tmp+=c*in[i+1];
5      out[i]=tmp;
6  }
```

Listing 2.2: Pseudocode for 1D 3-Point Jacobian Stencil, ILP=4

```
1  for(int i=0; i<N; i+=4){
2      for(int ilp=0; ilp<4; ilp++){
3          tmp[ilp]=a*in[i-1+ilp*4];}
4      for(int ilp=0; ilp<4; ilp++){
5          tmp[ilp]+=b*in[i+ilp*4];}
6      for(int ilp=0; ilp<4; ilp++){
7          tmp[ilp]+=c*in[i+1+ilp*4];}
8      for(int ilp=0; ilp<4; ilp++){
9          out[i+ilp*4]=tmp[ilp];}
10 }
```

2.3.2. Occupancy

Occupancy is a special concept in CUDA programming [64]. It is the ratio of activated warps to the maximum warps a GPU can host. Occupancy can also act as a measurement of the level of TLP.

2.3.3. Instruction Level Parallelism (ILP)

We adopt the same concept used in paper [63]: Instructions that can run independently. As an example, Listing 2.1 shows a naive 1D 3-Point Jacobian Stencil implementation. Intuitively, line 4-6 depends on the previous code so $ILP = 1$. Listing 2.2 shows a 1D 3-Point Jacobian Stencil implementation with $ILP = 4$ because instructions inside the loops on line 3 (also in line 7, 11, and 15, respectively) can process independently.

2.3.4. Wrapper Up

We consider two ways of providing parallelism: the number of threads (Thread Level Parallelism, TLP) and Instruction Level Parallelism (ILP). So, we have:

$$\mathbb{PAR} = TLP \times ILP \tag{2.8}$$

2.4. Minimum Necessary Parallelism

This section follows the path of parameter balancing analysis summarized by Volkov [63, 65].

We consider that a code saturates the hardware when the parallelism $\mathbb{PAR}$ provided by the code exceeds the concurrency indicated by the hardware:

$$\mathbb{PAR} \geq \mathbb{C} \tag{2.9}$$

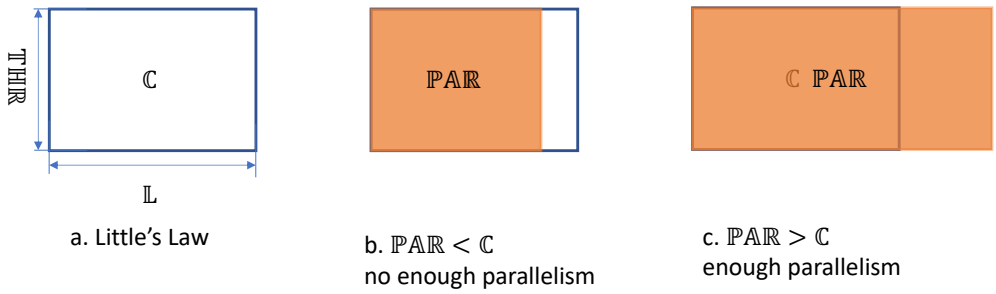


Figure 2.2: Minimal parallelism explanation: using $\mathbb{C}$ and $\mathbb{PAR}$ to determine whether the device is saturated.

Contrary to Volkov's analysis [65], which aims to maximize parallelism with a combination of ILP and TLP, our objective is to identify a minimal combination of TLP and ILP that can still saturates the device:

$$\begin{aligned} \underset{TLP,ILP}{\text{minimize}} \quad & \mathbb{PAR}(TLP, ILP) \\ \text{subject to} \quad & \mathbb{PAR} \geq \mathbb{C} \\ & \mathbb{PAR} = TLP \times ILP \\ & \mathbb{C} = \mathbb{L} \times \mathbb{THR} \end{aligned}$$

Figure 2.2 illustrates how concurrency $\mathbb{C}$ and parallelism $\mathbb{PAR}$ can be used to determine whether the device is saturated.

2.5. CUDA Programming Model and Execution Model

CUDA is a C-like programming model for Nvidia GPUs. It offers three levels of programming abstractions: *thread*, *thread block*, and *grid*.

thread is the most basic programming abstraction. At the hardware side, there is a hierarchy that maps to the CUDA programming model.

warp in CUDA is a small number of threads executed together as a working unit in a SIMT fashion. A warp in all Nvidia GPU generations consists of 32 threads. Inside an SM in V100 there are 4 warp schedulers corresponding to the 4 partitions inside one SM.

thread block and *grid* are higher programming abstractions built on top of *thread*. CUDA's Execution Model will map one thread block to only one SM, and one grid to only one GPU.

Figure 2.3 shows the details of CUDA programming model, its corresponding hardware abstraction, and the mapping relationship between them.

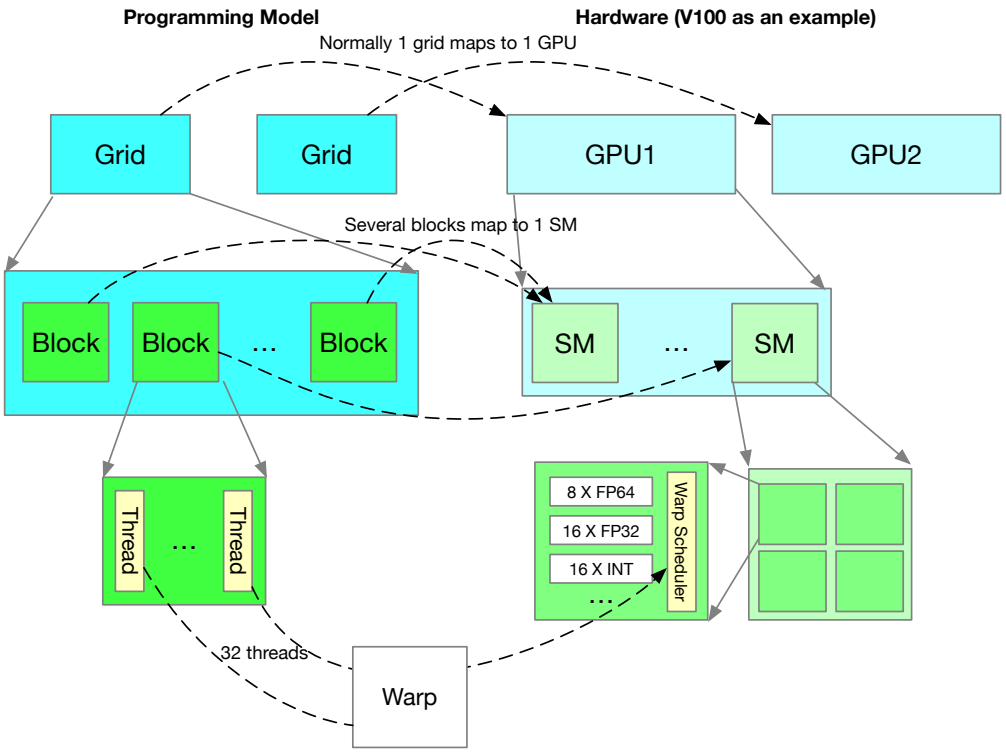


Figure 2.3: CUDA programming model and execution model that map the programming abstractions to hardware components

2.6. Overview of Synchronization Methods in Nvidia GPUs

2.6.1. Primitive Synchronization Methods in Nvidia GPUs

Starting from CUDA 9.0, Nvidia added the feature of *Cooperative Groups (CG)*. This feature is planned to allow scalable cooperation among groups of threads and provide flexible parallel decomposition. Coalesced groups and tile groups can be used as a method to decompose thread blocks. Beyond the level of thread blocks, grid synchronization is proposed for inter-block synchronization. Multi-grid synchronization is proposed for inter-GPU synchronization.

In the current version of CUDA (11.5), tile group and coalesced group only work correctly inside a warp. Analysis of PTX code shows that those two instructions are transformed to the *warp.sync* instruction. Hence, as it stands, we consider the synchronization capability of those methods to be only applicable to the warp level.

Figure 2.4 shows the granularity of cooperative groups and synchronization in the current version of CUDA.

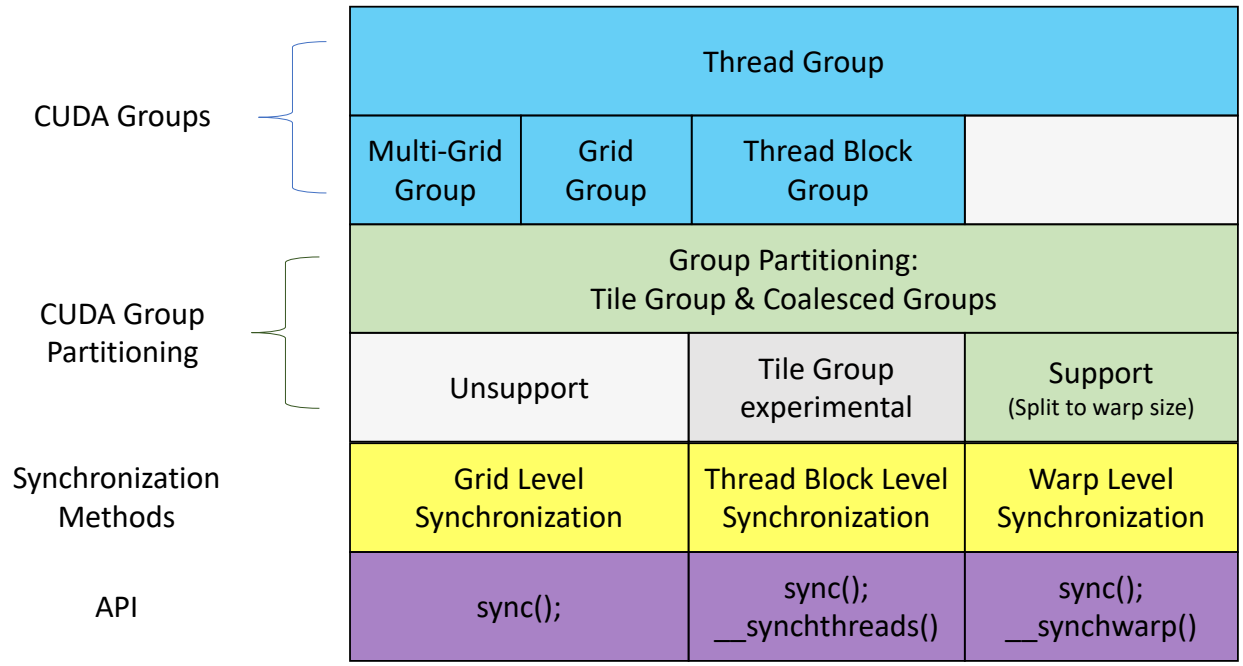


Figure 2.4: Hierarchy of synchronizations in CUDA

Warp Level Synchronization

Current CUDA supports two intra-warp synchronization methods, i.e. tile synchronization and the coalesced group synchronization corresponding respectively to the tile group and coalesced group[1] in Figure 2.4. Current versions of CUDA guarantee that all threads inside a warp process the same instruction at a time. Yet the introduction of synchronization methods inside a warp plus the fact that each thread now has its own Program Counter (PC) implies a future possibility of removing this feature.

Thread Block Level Synchronization

Thread block level synchronization corresponds to the thread block in the programming model. According to CUDA's programming guide [64], its function is the same as the classical synchronization primitive *__syncthreads()*.

Grid Level Synchronization

Starting from CUDA 9.0, Nvidia introduced grid group grid level synchronization. Grid level synchronization is a method to do single GPU synchronization. In order to use a grid group, cudaLaunchCooperativeKernel() API call is required, in comparison to the traditional kernel launch (<<<>>>).

[1]Even though both methods are derived from CUDA group partitioning APIs, currently these APIs not mature yet only statically support up to 32 splitting, (tiled group experimentally support up to 512 splitting. So we only consider them both warp level synchronization

Multi-Grid Level Synchronization (Multi-GPU Synchronization)

CUDA 9.0 also introduced the concept of multi-grid group. This group is initialized by a kernel launch API: cudaLaunchCooperativeKernelMultiDevice(). Synchronizing this group can do multi-GPU synchronization in a single node.

2.6.2. Non-primitive Synchronization

Software Barrier for Synchronization

Li etc. [66] researched fine-grained synchronization. Beyond it, Xiao, etc. [67] introduced a software device level synchronization. The authors limit the number of blocks per SM to only one in order to avoid deadlocks. Sorensen et al. extended this work by adding an automatic occupancy discovery protocol to discover activate warps [68].

Implicit Barrier for Synchronization

Before the introduction of grid level synchronization, the typical way to introduce a device-wide barrier to a program was to use several kernels in a single CUDA stream. A stream is a logical queue that enforces an execution order on the CUDA kernels in the stream, i.e. the kernels and data movement commands are executed in the order by which they appeared in the stream. For example, many DL frameworks, e.g., Chainer [69], use this method to enforce execution order.

Multi-GPU Synchronization

The common way to do multi-GPU synchronization is to synchronize CPU threads orchestrating the GPUs. The basic idea is to use one CPU thread per device (or one MPI rank per device). Additionally, with the help of the *GPUDirect* CUDA technology, it is also possible to implement multi-GPU software barriers using GPUDirect APIs.

2.7. GPU Memory Hierarchy

On-chip memory in a streaming multiprocessor (SMX) includes shared memory (scratchpad memory), L1 cache, register file (RF), and L2 cache. Off-chip memory includes global memory. Data in global memory can reside for the entirety of the program, while data in on-chip memory has the lifetime of a kernel. The shared memory is shared among all threads inside a thread block.

2.8. Iterative Algorithms

In iterative algorithms, the output of time step k is the input of time step $k + 1$. Iterative methods can be expressed as:

$$x^{k+1} = F(x^k) \tag{2.10}$$

When the domain is mapped out to processing elements, there are two points to consider:

- Spatial dependency necessitates synchronization between time steps, or else advancing the solution

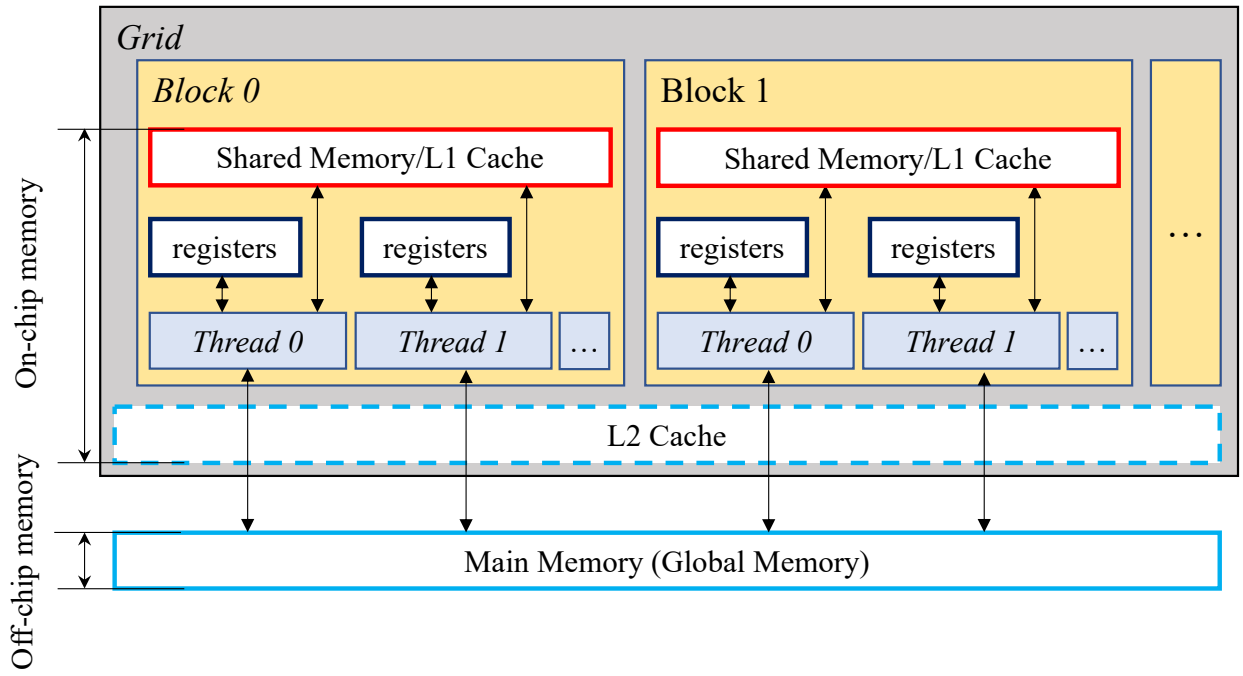


Figure 2.5: The relationship between CUDA Programming Model and GPU memory hierarchy. The italic format in color filled box marks CUDA Programming Model. Bank-filled box marks represent memory hierarchy.

in the following time step might use data that has not yet been updated in the previous time step.

- In time step $k+1$, each thread or thread block needs input from the output of itself in time step k (i.e. temporal dependency). This gives the opportunity for caching data between steps to reduce device memory traffic.

In the following sections, we briefly introduce iterative stencils and Krylov subspace methods. Throughout the paper, we use them as motivation examples, and we use them to report the effectiveness of our proposed methods, given their importance in HPC scientific and engineering codes.

2.8.1. Iterative Stencils

Iterative stencils are widely used in HPC. According to Bastian et al. [34], stencil applications represent 49% of workloads in a wide range of HPC centers. Take 2D Jacobian 5-point stencil (2d5pt) as an example:

$$\begin{aligned} x(i,j)^{k+1} =& N * x(i,j+1)^k + S * x(i,j-1)^k + \\ & C * x(i,j)^k + W * x(i-1,j)^k + E * x(i+1,j)^k \end{aligned} \tag{2.11}$$

Computation of each point at time step $k+1$ requires the values of the point itself and its four neighboring points at time step k.

Two blocking methods are widely used to optimize iterative stencils for data locality: *Spatial Blocking* [70, 71] and *Temporal Blocking* [2, 4, 5].

In spatial blocking on GPUs, we split the whole domain into sub-domains, where each thread block can load its sub-domain to the shared memory to improve data reuse. In the meantime, we require redundant data accesses at the boundary of the thread block to data designated for adjacent thread blocks.

In iterative stencils, each time step depends on the result of the previous time step. One could advance the solution by combining several time steps. The temporal dependency, in this case, is resolved by using a number of halo layers that match the number of combined steps. The amount of data that can be computed depends on the stencil radius (rad) and the number of time steps that are combined (b_t). In overlapped temporal tiling [72–74], this region can be represented as $2 \times b_t \times rad$ ($halo$ region). Methods based on this kind of blocking are called *overlapped temporal blocking* schemes. Overlapped temporal blocking introduced the overhead of redundant computation that wavefront [46, 48, 75] is aimed to alleviate.

2.8.2. Krylov Subspace Methods

Krylov methods are widely used for large sparse (and dense) linear systems of equations arising in solvers of Partial Differential Equations (PDEs) [76–78], as well as statistics, machine learning and control theory [79]. Krylov subspace methods can be described as:

$$\kappa_r(A, b) = span\{b, Ab, A^2, ..., A^{r-1}b\} \tag{2.12}$$

Assuming that A is an invertible matrix, it is possible to compute $x = A^{-1}b$ (or solve $Ax = b$) by searching the Krylov subspace without directly computing A^{-1}. Searching the Krylov subspace is a sequence of matrix-vector multiplications, where at each step the approximation of the solution vector x is updated proportionally to the residual error (vector r) from the previous time step.

Conjugate gradient is a main solver in the family of Krylov subspace methods. It is mainly used to solve systems of linear equations for symmetric and positive-definite matrices.

CHAPTER 3

Microbenchmarking Nvidia GPUs: A Decade of Evolution Trends

Over a decade passed since the first GPU-accelerated supercomputer TSUBAME [50]. While numerous advancements have been made since then, we have noticed two specific trends in the evolution of GPUs (with a focus on Nvidia GPUs [1]). First, the significant increase in the capacity of user-managed cache (shared memory), which escalated from 720 KB in the K20 [80] to 17,712 KB in the A100 [20]—an increase by a factor of 24.6 over recent decades. Second, we observe that the latest modules can easier saturate the device. In addition, GPUs provide features that have been supported by CPUs for years. Examples include cooperative groups (i.e., device-wide barriers), low(er) latency of operations, and asynchronous copy of shared memory (i.e., prefetching) [64].

In this Chapter, we characterize the lineage of Nvidia GPUs in the last decades. Specifically:

- We trace the evolution of performance characteristics of different features in Nvidia GPUs, taking into account the capacity of on-chip resources, concurrency analysis, and the performance characteristics of different synchronizations (Section 3.1).
- We introduce two strategies to address the observed machine imbalances, derived from the insights gathered throughout this chapter (Section 3.2).
- We introduce the microbenchmark employed in our study. To showcase the applicability of these synchronization microbenchmarks, we use the P100 and A100 as our research targets (Section 3.3 and Section 3.4). The implementations are available at https://github.com/neozhang307/SyncMicrobenchmark.
- We delve into the potential pitfalls when using several synchronization instructions (Section 3.5).
- We provide various implementations of the reduction operator as a motivating example to demonstrate how synchronization might influence the performance (Section 3.6). The implementations are available at https://github.com/neozhang307/Reduction.

3.1. GPU Evolution: Trends and Insights

In this section, we summarize the statistics of interest for this dissertation. We gather the majority of data from Nvidia documents [20, 21, 36, 80]. For undocumented data, we use microbenchmarks (Section 3.3) to collect necessary data.

[1] We focus on Nvidia GPUs in this research due to the continuity of Nvidia's GPU products over the years, which provides a solid grounds for observing changes.

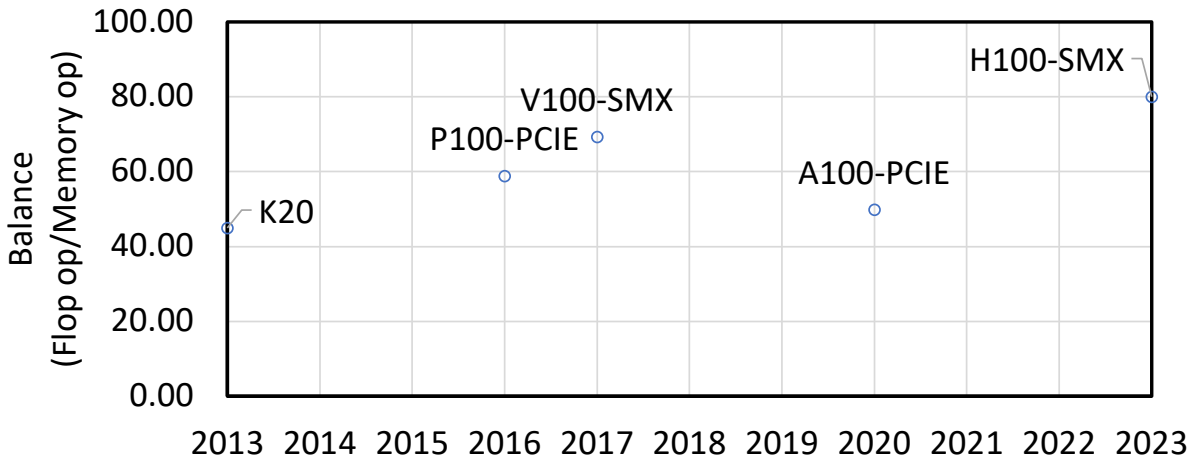


Figure 3.1: The trend of balance in latest data center GPUs

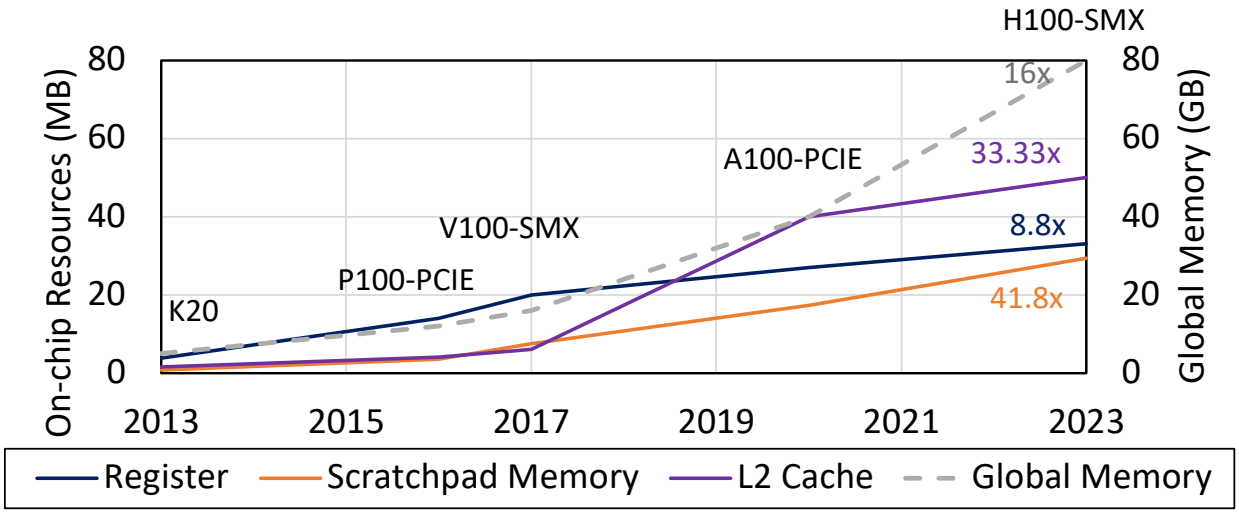


Figure 3.2: The capacity trend of memory in latest data center GPUs

3.1.1. Machine Balance

Figure 3.1 portrays the machine balance (Section 2.1.1) of Nvidia GPUs over the past decade. The A100-PCIE platform is relatively balanced but needs 50 times double precision operation to overlap one memory access operation. More concerning is the upcoming H100-SMX, which appears to exacerbate this imbalance further.

3.1.2. On-chip Resources

Figure 3.2 illustrates the memory resources statistics (from global memory to register files) of Nvidia GPUs over the last decade. In general, all memory resources are increasing over the last decade. Notably, the capacity of cache, both scratchpad memory and L2 cache, is increasing at a faster pace. As such, if this trend continues in the next decade, we can anticipate that both the

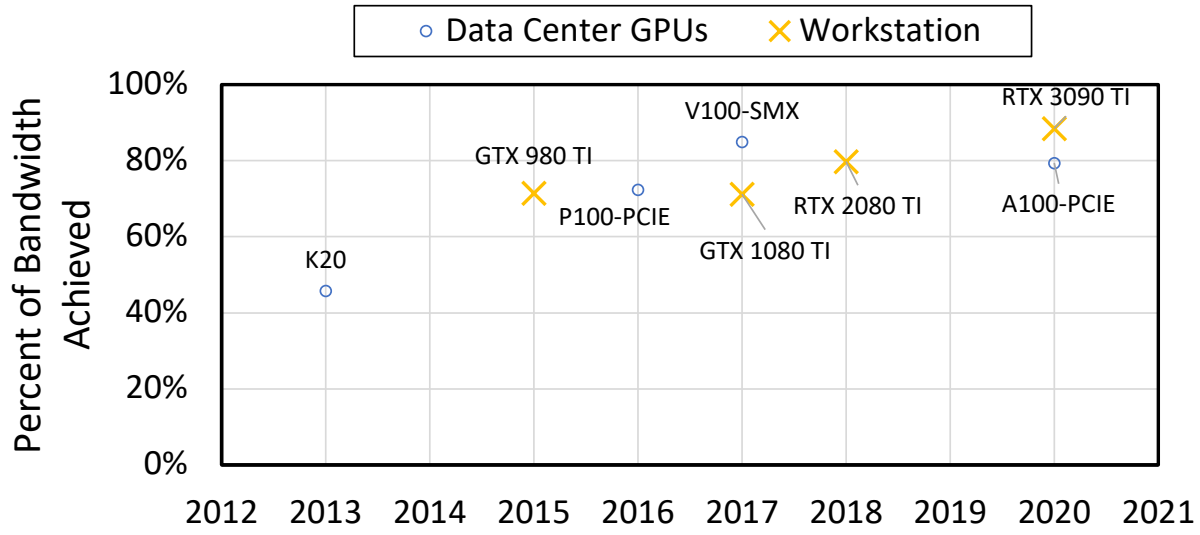


Figure 3.3: Low occupancy (occupancy=256, ILP=4, 8 Byte per memory access) STREAM kernel performance in latest data center GPUs

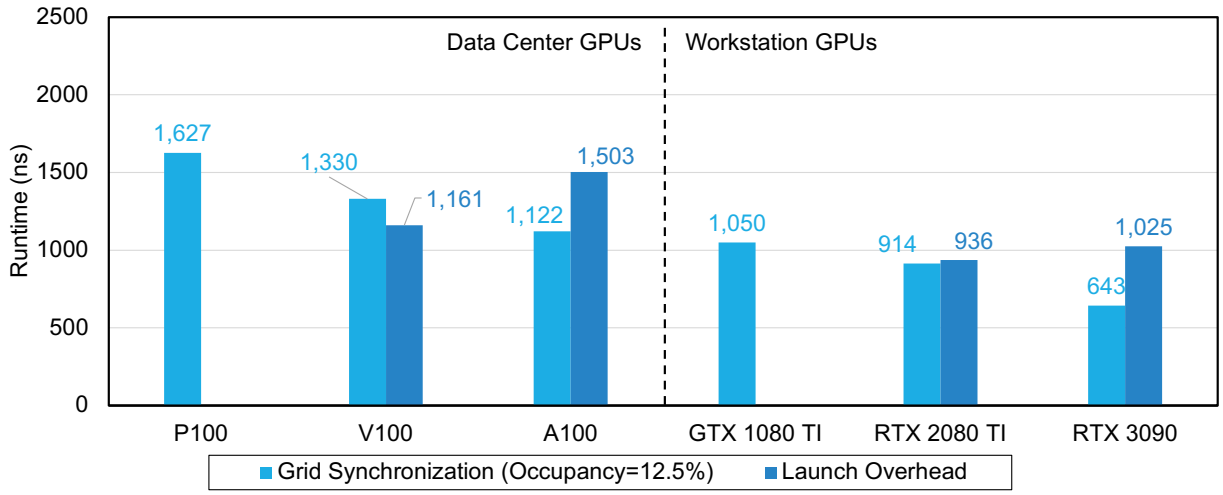


Figure 3.4: Overhead of device-wide synchronization (grid synchronization as explicit device-wide synchronization, kernel launch as implicit device-wide synchronization) in the latest GPUs

capacity of L2 cache and L1 cache can exceed one Gigabyte.

3.1.3. Easier Memory Bus Saturation

Section 3.1.1 highlighted the imbalance in the latest GPUs. This imbalance renders many kernels memory-bound. Thus global memory access becomes a critical component. In this section, under the restriction of low occupancy, we examine the performance of global memory access over a decade. Please note that the H100 model has been excluded from our analysis due to our lack of access.

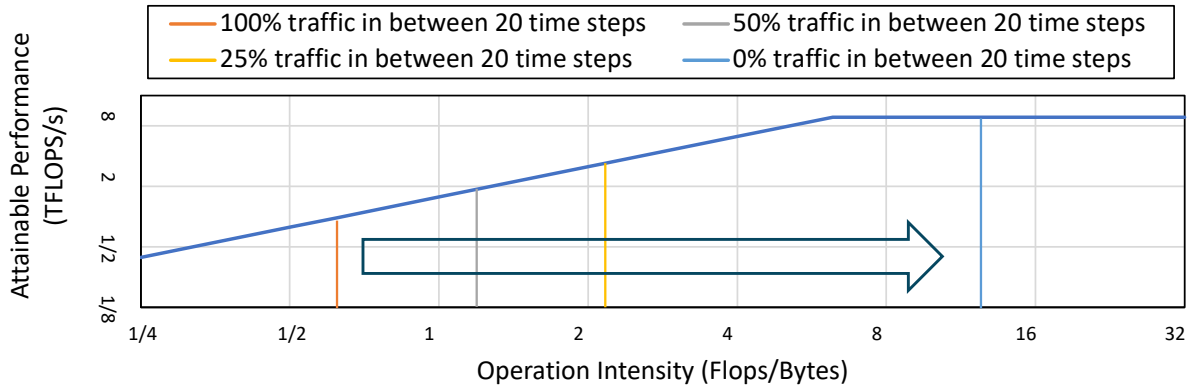


Figure 3.5: The effect of combining 20-time step and reducing memory traffic in between time steps, 2D 5-Point Jacobian Stencil in A100 as an example

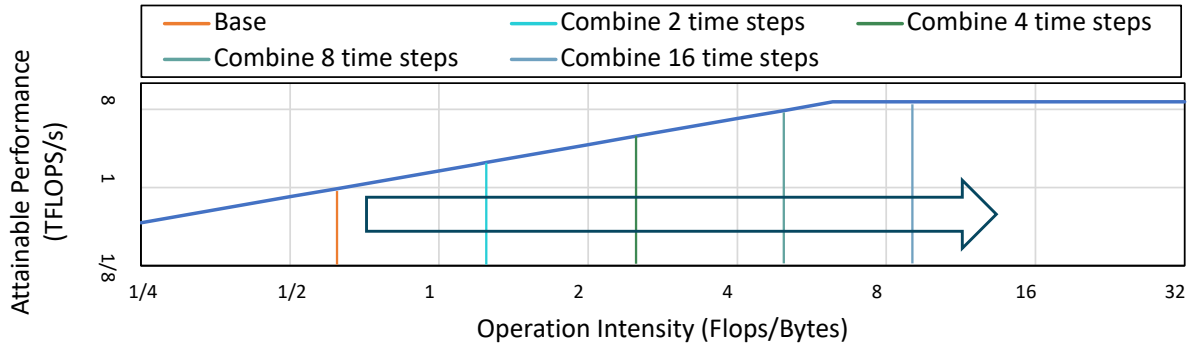


Figure 3.6: The effect of combining time steps and eliminating memory traffic in between the combined time steps, 2D 5-Point Jacobian Stencil in A100 as an example

According to Little's Law, theoretically, a configuration comprising a thread/stream multiprocessor (TLP) setting of 256 and Instruction Level Parallelism (ILP) of 4 with double precision (8 Bytes) per access can saturate all the devices we investigated. We implement a simple STREAM kernel $A[] = B[]$ using this configuration. We plot the proportion of peak performance achieved with this STREAM kernel in Figure 3.3.

As Figure 3.3 shows, with occupancy=256 (TLP=256), achieving 80% of theoretical bandwidth in the latest GPUs is possible. However, the GPU ten years ago (K20), can hardly achieve 50% of theoretical bandwidth with such a setting.

3.1.4. Grid Level Synchronization

In Figure 3.4, we illustrate the overheads of device-wide synchronization (Section 2.1.1) for the latest Nvidia GPUs. We can see that the overhead of grid-level synchronization is decreasing upon a new version of machines.

3.2. Strategies to Overcome the Machine Imbalance

3.2.1. Strategy 1: Combine Time Steps

A viable strategy to enhance data locality is extending the kernel's lifetime across time steps, leveraging the vast register files and scratchpad memory to *reduce* or *eliminate* device memory traffic in between time steps.

Consider a 2D 5-Point Jacobian Stencil in an A100 GPU as an example. The basic kernel's Operation Intensity is 5/8. Given the A100's balance of 6.24, this given kernel is far from fully utilizing the A100's computational power.

In the first scenario, we combine 20 time steps to *reduce* memory traffic to the global memory between time steps. Figure 3.5 illustrates the effect of decreasing memory traffic between time steps. As more memory traffic is reduced between time steps, operation intensity heightens.

In the second scenario, we combine time steps to *eliminate* memory traffic to the global memory in between time steps. Figure 3.6 showcases the effect of integrating time steps. As more time steps are combined, the kernel's operation intensity escalates.

Both methods allow for an increase in operation intensity without necessitating the algorithm's redesign.

3.2.2. Strategy 2: Minimal Parallelism

Insights derived from microbenchmarking reveal that the latest GPU platforms can sustain lower occupancy levels. Accordingly, we propose a strategy of minimal parallelism. This maximizes on-chip resources per thread (maximizing the effect of the first strategy) and minimizes synchronization overheads (as the following section will demonstrate).

3.3. Microbenchmarks for GPUs

3.3.1. Microbenchmarks for Kernel Launch Overhead (Implicit Barriers)

Launching new kernels in a single stream can act as a device-wide implicit barrier to maintain the order of the program. Yet launching an additional kernel is not a free lunch: it also introduces overheads. This section will inspect the overhead of kernel launch, including traditional launch function, i.e., the <<<>>> kernel invocation method, and the new launch functions, i.e., cudaLaunchCooperativeKernel() and cudaLaunchCooperativeKernelMultiDevice() Nvidia introduced from CUDA 9.0 for Cooperative Launch (necessary for grid level or multi-grid level synchronization).

We do not consider the extra overhead of launching the first kernel to simplify our discussion. Instead, in all our measurements, we assume a warm-up kernel already launched, and we focus our analysis on the behavior of kernels launched after the warm-up kernel.

We use sleep instruction available after Volta architecture to control the kernel execution time.

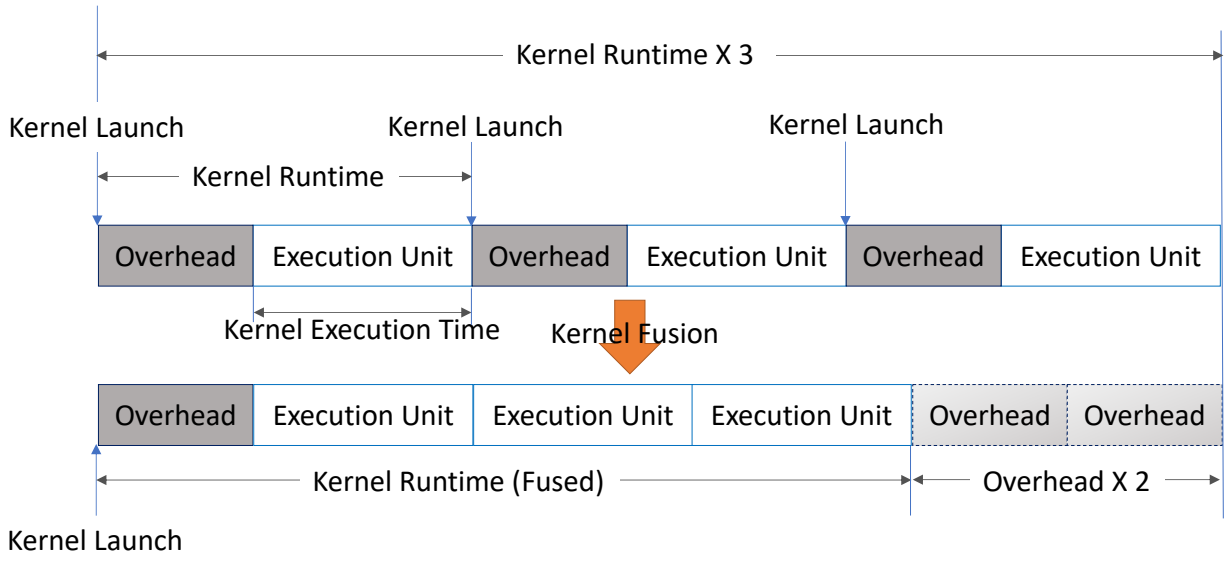


Figure 3.7: Elaborate the methodology for testing implicit barrier overhead

Firstly, we focus on kernels with execution time longer than 10 ns (longer than the total latency of an empty kernel).

Before further discussion in this section, we introduce the following terms:

- **Kernel Execution Time:** Total time spent executing the kernel, excluding any overhead for launching the kernel (controlled by sleep instruction).
- **Launch Overhead:** Latency unrelated to kernel execution.
- **Kernel Runtime:** Total runtime of a kernels

$$T_{Kernel\ Runtime} = T_{Kernel\ Execution\ Time} + T_{Launch\ Overhead} \tag{3.1}$$

Figure 3.7 further explains the relationship between Kernel Execution Time, Kernel Runtime, and Launch Overhead (marked as Overhead).

Kernel Execution Time

Before Diving into details, we conducted a fast check to figure out the necessary Kernel Execution Time by assuming that nanosleep instruction can set precisely 1000 ns. Based on Equation 3.2, we control Kernel Execution Time to be ranged from 0 us to 8 us. The figure showed the relationship between Kernel Execution Time and Kernel Runtime and inferred Launch Overhead. Noting that when Kernel Execution Time is small, the overhead might be higher. To understand the Launch Overhead in a practical situation, when a single kernel usually last longer than 2 us, we use the sleep instruction introduced in CUDA for the Volta platform.

It is worth mentioning that nanosleep instruction could not accurately control kernel execution

Listing 3.1: Sample code to micro-benchmark implicit barriers for a kernel with nanosleep instruction.

```
__global__ void sleep_kernel(){
    //kernel execution latency is 10 us here.
    repeat10(asm volatile("nanosleep.u32 1000;"););
}
...
record(timer1);
repeat1(launch(sleep_kernel, launchparameters););
cudaDeviceSynchronize();
record(timer2);
repeat5(launch(sleep_kernel, launchparameters););
cudaDeviceSynchronize();
record(timer3);
...
```

time. So in the following analysis, we do not assume the exact Kernel Execution Time produced by nanosleep instruction.

Measuring Kernel Runtime

Listing 3.1 is our sample code for micro-benchmarks. We acquires the Kernel Runtime with the following equation:

$$T_{Kerne\ Runtime} = ((timer3 - timer2) - (timer2 - timer1))/(5 - 1) \tag{3.2}$$

In this way we eliminate the effect of warmup kernels.

Deducing Kernel Overhead with Kernel Fusion

The Overhead (O) can be deduced from $Latency_{ij}$ where i represents call launch function i times and j represents launch kernel with j execution unit (1 us).

$$O = \frac{Latency_{ij} - Latency_{ji}}{i - j} \tag{3.3}$$

We use kernel fusion to unveil the overhead hidden in kernel runtime. The basic assumption here is that: 1) merging the work of multiple argument-less kernels into one single kernel does not introduce additional launch overhead, and 2) the time saved when using kernel fusion should be equal to the overhead of launching an additional kernel. From our previous observations, sleep instruction has insignificant overhead and fits well into this assumption. In this situation, we compute the overhead with Equation 3.3.

3.3.2. Microbenchmark for Intra Stream Multiprocessors Instructions

We directly use Wong's [81] method for instruction micro-benchmarking. Wong's method relies on the GPU clock. The basic methodology is to build a chain of dependent operations to repeat a single instruction enough times to saturate the instruction pipeline. By using the clock register to record the begin and the end timestamps of the series of operations, it is possible to average the

Listing 3.2: Sample code to measure the latency of the add instruction in GPU.

```
__global__ void kernel1(){
    start=clock();
    repeat256(p=p+q;q=p+q); //total repeat=512
    end=clock();
    return q;
}
...
__global__ void kernel2(){
    start=clock();
    repeat512(p=p+q;q=p+q); //total repeat=1024
    end=clock();
    return q;
}
...
cpuclock();
kernel();
syncdevice();
cpuclock();
```

repetitions to infer the latency of that instruction. Listing 3.2 shows an example sample code to measure the latency of an add instruction.

3.3.3. Microbenchmark for Inter Stream Multiprocessor Instructions

Jia's work [82] can work correctly only inside a single thread, Wong's work [81] can work correctly only in a single Stream Multiprocessor. Yet current synchronization instructions might involve cooperation across different threads, Stream Multiprocessors, and GPUs. As we move to grid level synchronization and beyond, we need a new method. Specifically, a global clock is necessary to test the performance of synchronization beyond a single Stream Multiprocessor. In CUDA's execution model, a CPU thread launches a kernel, and it can call the DeviceSynchronize() function to block the CPU thread until the GPU kernel finishes execution. So it is possible to use the clock in the CPU thread as a global clock to test GPU instructions. Yet it introduces two issues:

- We need to eliminate any latency unrelated to the target instruction
- Account for the relative inaccuracy in the CPU clock measurement compared to the GPU's clock measurement.

To solve those issues, we introduce a new microbenchmark method. If we increase the repetitions of instructions in the GPU kernel (in Listing 3.2), the additional kernel latency is only related to the additional repeat times of instructions. In this manner, we can avoid unrelated latency from kernel launch (to get more accurate measurements). Equation 3.4 shows how to measure the instruction latency with this method (first issue solved).

$$T_{instruction} = \frac{T_{k_1} - T_{k_2}}{r_{k_1} - r_{k_2}} \tag{3.4}$$

Standard deviation can represent the uncertainty in a single measurement [83]. We can infer

the error rate σ from kernel total runtime T and the workload repeat times in kernel i (r_{k_i}):

$$\begin{aligned}
\sigma_{\frac{k_1-k_2}{r_{k_1}-r_{k_2}}} &= \sqrt{\frac{\sum_{n=1}^{N}\left(\frac{T_{k_1}-T_{k_2}}{r_{k_1}-r_{k_2}}\right)^2 - \sum_{n=1}^{N}\left(\frac{\overline{T_{k_1}-T_{k_2}}}{r_{k_1}-r_{k_2}}\right)^2}{N-1}} \\
&= \frac{1}{r_{k_1}-r_{k_2}}\sqrt{\frac{\sum T_{k_1}^2 - \overline{T_{k_1}}^2}{N-1} + \frac{\sum T_{k_2}^2 - \overline{T_{k_2}}^2}{N-1}} \\
&= \frac{1}{r_{k_1}-r_{k_2}}\sqrt{\sigma_{k_1}^2 + \sigma_{k_2}^2}
\end{aligned} \tag{3.5}$$

As Equation 3.5 shows the standard deviation of the instruction tested and its deduction (the measurement of kernel 1 and kernel 2 is independent of each other). And by deduction, if the difference in repeat times is large enough, the standard deviation of the instruction latency we seek to measure will be small (second issue solved).

We use Wong's and our methods to test the single precision *ADD* instruction to verify that the proposed method is feasible. Both results show that float-add costs 6 cycles in P100 and 4 cycles in V100. Those results match the result in [82]. We can conclude that the inter SM micro-benchmark method we propose is a reliable measurement tool that approaches the accuracy of the GPU clock.

We verify that the repeat times of a synchronization instruction would not influence the thread block and grid level performance. Tile shuffle at warp level also works as we anticipated. Other warp level synchronization can be unstable: the latency of the synchronization instruction might increase suddenly when increasing repeat times. It could be the case that this warp synchronization relies on software implementation. So when repeating an instruction too many times, instruction cache overflow can occur. We only record the fastest result for warp level synchronization instructions.

3.4. Microbenchmarking GPU Synchronization Instructions

3.4.1. Microbenchmarking Single-GPU Implicit Barriers

This subsection presents the result of a single GPU implicit barrier overhead. The result of the multi-GPU implicit barrier is presented in Section 3.4.3.

We measured the launch overhead by using the kernel fusion method we introduced in Section 3.3.1. We also test the kernel total runtime of a null kernel for comparison. Table 3.1 shows the result. Generally, Implicit Barrier Overhead is at the *us* level.

To the best of the authors' knowledge, Volkov et al. [84] was the first to measure the implicit barrier's overhead, i.e., CUDA kernel launch overhead. Xiao et al. [67] additionally build a model for implicit and explicit barriers. They both neglect that the launch overhead is far smaller after launching the warmup kernel. When kernel execution time (representing the workload) is long enough, the launch overhead would be less. When using null kernels, we tested a launch overhead of around $3us$ for a traditional launch, which is close to the best case reported by Volkov et al. [84].

Table 3.1: Launch Overhead and Null Kernel Runtime of Different Launch APIs

Launch Type	Launch Overhead (ns)	Empty Kernel Kernel Runtime (ns)
Traditional	1081	8888
Cooperative	1063	10248
Cooperative Multi-Device	1258	10874

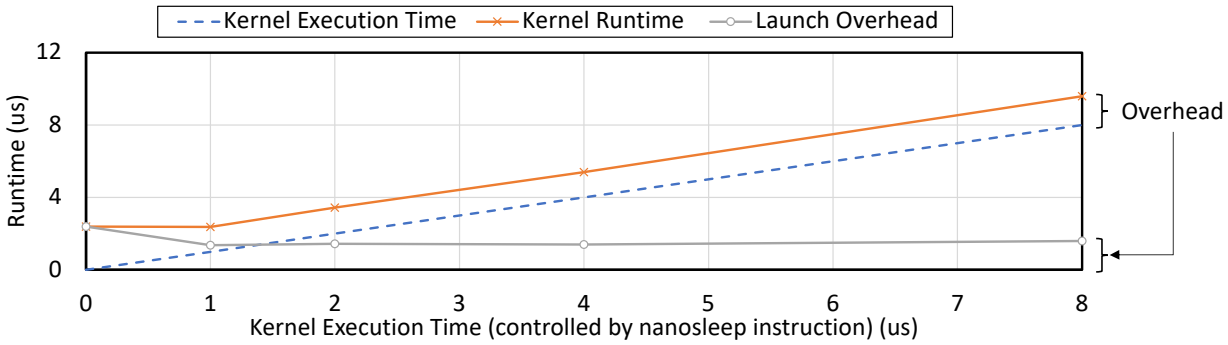


Figure 3.8: How Kernel Execution Time influences Kernel Runtime and inferred Launch Overhead

We plot how Kernel Execution Time might result in different kernel launch overheads in Figure 3.8.

3.4.2. Microbenchmarking Single-GPU Explicit Barrier

For the warp shuffle operation and block synchronization operation, the throughput is reported by CUDA programming guide [64] at the granularity of warps and thread blocks, respectively. However, it's possible that the size of the group conducting synchronization or shuffle could impact performance. Hence in this work, we consider the group size when experimenting with warp shuffle and block synchronization.

Warp-Level Synchronization. The current CUDA (10.0) supports two kinds of warp level synchronization: tile group based and coalesced group based (as seen in Figure 2.4). Additionally, the CUDA shuffle operation, which exchanges a register value among threads in a warp, is an operation that implies synchronization after it. We thereby also include the results of the shuffle operation.

Since the size of a synchronization group might influence the result, we tested every possible group size for both tile group and coalesced group. The possible tile group sizes are: 1, 2, 4, 8, 16, and 32. The possible coalesced group size ranges from 1 to 32. Latency is tested using only 32 threads (a warp) in a CUDA kernel with one thread block. The throughput is tested by iterating every possibility pair of up to 1024 threads and up to 64 thread blocks per Stream Multiprocessor and recording only the highest result. Table 3.2 shows the result of warp level synchronization.

For tile group synchronization the size of the group influences neither latency nor throughput. A possible explanation is that CUDA might merge all the concurrent tile group synchronization

Table 3.2: Performance of Warp Synchronization in a Thread Block

Type (group size)	**Latency cycle**		**Throughput (sync/cycle)**		**Reference** [64] thread op/cycle	
	V100	**P100**	**V100**	**P100**	**V100**	**P100**
Tile(*)	14	1	0.812	1.774	-	-
Shuffle(Tile)(*)	22	31	0.928	0.642	32	32
Coalesced(1-31)	108	1	0.167	1.791	-	-
Coalesced(32)	14	1	1.306	1.821	-	-
Shuffle(COA)(*)	77	50	0.121	0.166	-	-
Thread Block(warp))	22	218	0.475	0.091	16	32

instructions into a single one. For coalesced group synchronization, the group size does not influence the performance of P100. The group size does, however, influence the performance of the coalesced group in V100. The performance is the highest when all the threads inside a warp belong to a single coalesced group. For convenience, because the group size doesn't influence the total latency of tile group synchronization, we only record the throughput for a group size of 32 in tile group synchronization.

We include the reference throughput of shuffle operation mentioned in the CUDA programming guide [64] in Table 3.2. The performance of V100 is close to the theoretical result in the programming guide. Conversely, it appears that some overheads are affecting the throughput of the shuffle operation in P100.

Thread Block Level Synchronization We tested every possible group size at the thread block level, i.e., ranging from 32 to 1024. We find that the throughput of block level synchronization is related to the number of active warps per Stream Multiprocessor.

Figure 3.9 shows the relationship between the throughput of thread block synchronization divided by warp count (warp sync per cycle) and the maximum number of activate warps per Stream Multiprocessor (as calculated by [36]). When the warp count exceeds the size of the max activated warp per Stream Multiprocessor, the device is saturated, and the throughput of thread block synchronization reaches its maximum. The throughput of block synchronization is less related to the launch parameters of thread per thread block or thread block per stream multiprocessors individually.

With this observation, we conclude that the performance of thread block level synchronization is related to the warp count per Stream Multiprocessor. We further summarize the performance of thread block synchronization from a warp's perspective in Table 3.2.

CUDA's programming guide [64] reports that the throughput for *__syncthreads()* (or block-level synchronization) is 16 operations per clock cycle for capability 7.x (V100) and 32 for capability 6.0 (P100). The throughput of V100 is relatively close to 16 op/cycle. But the result of P100 is far

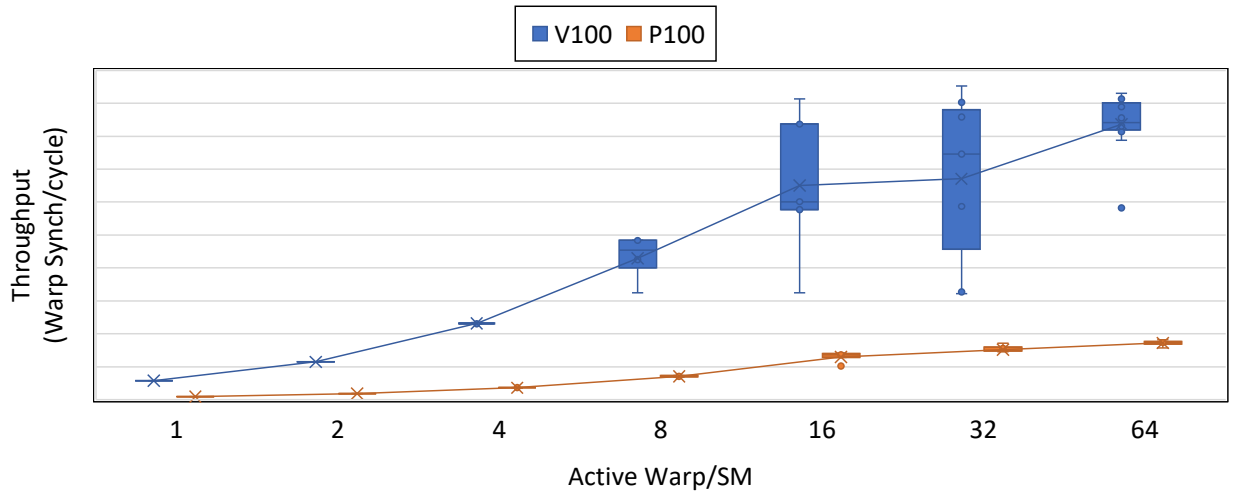


Figure 3.9: Relationship between throughput of block sync and active warp/SM (warp per Stream Multiprocessor)

from 32 op/cycle.

Admittedly, it is also possible that the performance of thread block synchronization in P100 is not ideal due to over-subscription. Yet the latency of thread block synchronization in P100 is so large that it is nearly impossible to find a point at which the instruction pipeline is saturated while the overhead of over-subscription is not so severe.

Grid Level Synchronization Figure 3.10 shows the heat map of grid synchronization. It shows that in both V100 and P100, the latency of grid synchronization is more related to the grid dimension (specifically, thread block count per Stream Multiprocessor) than the thread block dimension.

It seems grid-level synchronization is at the same level as the overhead of kernel launch we measured in Section 3.4.1. So, single GPU grid synchronization might not benefit performance compared to implicit barrier methods. Yet we argue that this performance difference is negligible (at most $2.5us$ with 2 $block/SM$ (two thread blocks per Stream Multiprocessor)) in real applications. In addition, using the implicit barrier instead would eliminate the possibility of data reuse in shared memory and registers.

3.4.3. Microbenchmarking Multi-GPU Explicit Barrier

We consider three ways to do multi-GPU synchronization:

Multi-Device Launch Function as an Implicit Barrier

When using the multi-device launch function with the default flag, kernels will not execute until all the previous operations in all the GPU streams involved have finished execution [85]. Although this implicit barrier method is not commonly used, we evaluate it to assess if it is a valuable alternative. Section 3.4.1 discusses the micro-benchmark used in this method.

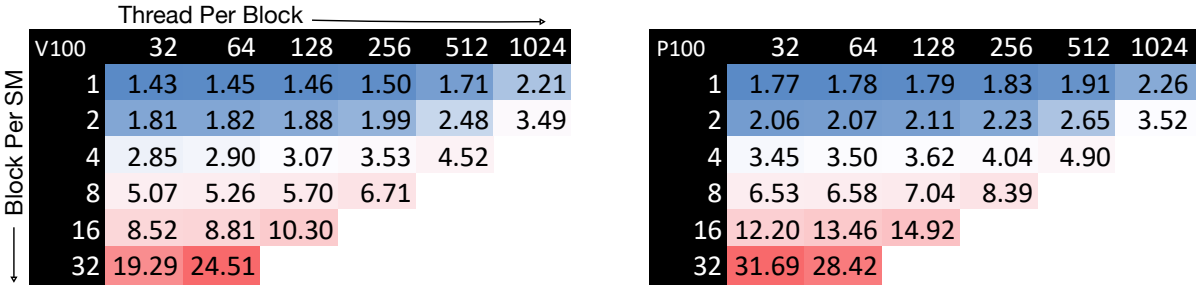


Figure 3.10: Latency (*us*) of grid synchronization in V100 (left) and P100 (right). In each chart, the number of thread blocks per Stream Multiprocessor (Block Per SM) increases vertically, while the number of threads per thread block (Thread Per Block) increases horizontally.

Listing 3.3: Code example of using CPU threads for synchronization.

```
#pragma omp parallel num_threads(GPU_count)
{
    unit gid=omp_get_thread_num();
    cudaSetDevice(gid);
    ...
    kernel<<<>>>();
    cudaDeviceSynchronize();
    #pragma omp barrier
    ...
}
```

CPU-Side Barriers

A common way to make a barrier between GPUs is to use CPU threads or processes to synchronize different GPUs. We use openMP to measure the overhead in this case. Each thread calls the cudaDeviceSynchronize() API to ensure the asynchronously launched GPU kernels are executed till their end. In addition, the threads use the openMP barrier API to synchronize. Listing 3.3 shows the code example for this barrier. Finally, we appropriately pin the CPU threads. We applied the same micro-benchmark discussed in Section 3.4.1 for this method.

Multi-Grid Synchronization

We used Inter-SM Instruction Microbenchmarks discussed in Section 3.3.1 for this method. Figure 3.11 and Figure 3.12 show the heat maps of the latency of multi-grid synchronization in V100 and P100. Because the interconnection in the P100 system is PCIe, the performance is worse than the V100 system equipped with NVLink connections between devices.

We experimented with all 8 GPUs in the DGX-1, and we found that the performance of multi-grid synchronization among 2-5 GPUs is similar to each other, and the performance of multi-grid synchronization among 6-8 GPUs is similar to each other. This behavior is likely related to the internal NVLink network structure of DGX-1. From Figures 3.11 and 3.12, we can see that the performance of multi-grid synchronization is influenced by both the grid dimension and the

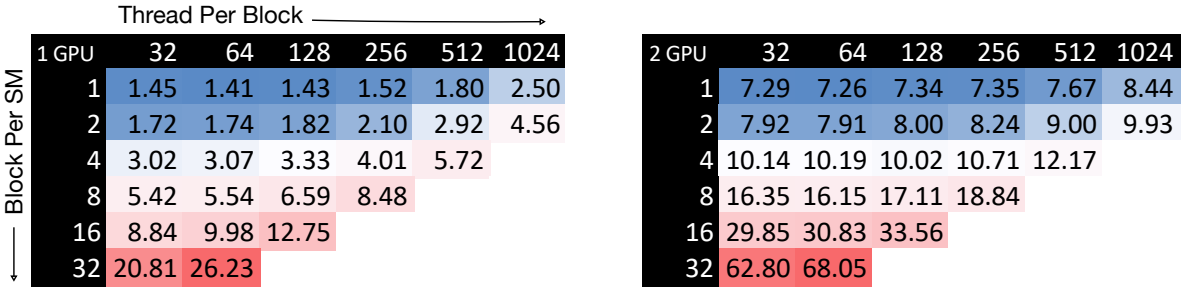


Figure 3.11: Latency (us) of multi-grid synchronization in P100 platform for one GPU (left) and two GPUs. In each chart, the number of thread blocks per Stream Multiprocessor (Block Per SM) increases vertically, while the number of threads per thread block (Thread Per Block) increases horizontally.

number of active warps per SM. The performance is acceptable when the thread block per Stream Multiprocessor is within 8 ($block/SM <= 8$) and warp per Stream Multiprocessor is within 32 ($warp/SM <= 32$). Apart from the case of one GPU, latency in all cases is no more than 2x slower than the fastest case, 1 thread block per Stream Multiprocessor and 32 thread per thread block setting (1 $block/SM$, 32 $threads/block$), and 2x faster than the slowest case, 32 thread block per Stream Multiprocessor and 64 thread per thread block setting (32 $blocks/SM$, 64 $threads/block$).

Comparison of Different Multi-GPU Synchronization Methods

Figure 3.13 shows the results of all three multi-GPU synchronization methods across 8 GPUs in DGX-1. For simplification, we only plot the data of three cases of multi-grid synchronization in Figure 3.13:

1. one block per Stream Multiprocessor and 32 thread per thread blocks (1 $block/SM$, 32 $threads/block$) as the fastest case,
2. 32 blocks per Stream Multiprocessor and 64 thread per thread blocks (32 $blocks/SM$, 64 $threads/block$) as the slowest case
3. one block per Stream Multiprocessor and 1024 thread per thread blocks (1 $block/SM$, 1024 $threads/block$) as a general case, which is within the parameters we recommended in the previous paragraph.

The CPU-side barrier relying on openMP barriers outperforms implicit barriers in multi-device launches when the GPU count is larger than two. Also, the overhead of the CPU-side barrier is relatively steady w.r.t. GPU count. It is worth mentioning that this result is close to the kernel total runtime of an empty kernel, as shown in Table 3.1.

Figure 3.13 shows two performance drops in multi-grid synchronization. We anticipated that the second drop would be between 4 GPUs and 5 GPUs, based on the internal network structure of DGX-1 that groups 4 GPU together. However, we find no reason for the performance drop between

Thread Per Block →

Block Per SM ↓

1 GPU	32	64	128	256	512	1024
1	1.42	1.44	1.56	2.04	3.06	7.34
2	1.81	1.86	2.33	3.34	6.93	18.97
4	2.92	3.37	4.35	7.53	19.10	
8	5.32	6.35	9.10	20.68		
16	9.66	11.72	24.24			
32	20.84	34.04				

2 GPU	32	64	128	256	512	1024
1	6.44	6.46	6.53	6.99	8.05	12.41
2	6.77	6.80	7.28	8.32	11.80	24.14
4	7.96	8.41	9.46	12.57	24.21	
8	12.47	13.63	16.55	28.03		
16	22.48	24.64	37.04			
32	45.88	58.60				

5 GPU	32	64	128	256	512	1024
1	7.02	7.05	7.15	7.62	8.68	13.32
2	7.37	7.44	7.92	9.01	12.72	25.16
4	8.61	9.14	10.14	13.41	25.23	
8	13.19	14.21	17.16	28.71		
16	23.58	25.61	38.15			
32	48.71	61.66				

6 GPU	32	64	128	256	512	1024
1	18.67	18.66	18.68	19.26	20.28	24.78
2	19.03	19.12	19.54	20.54	23.64	35.89
4	20.29	20.88	21.80	24.77	36.37	
8	23.39	24.43	27.18	38.93		
16	29.27	31.41	44.37			
32	54.24	69.70				

8 GPU	32	64	128	256	512	1024
1	20.97	21.00	21.10	21.42	22.55	26.93
2	21.18	21.41	21.85	22.81	25.98	37.99
4	22.62	23.04	24.13	27.08	38.60	
8	25.98	26.62	29.33	40.86		
16	32.20	33.67	45.98			
32	58.30	71.90				

Figure 3.12: Latency (*us*) of multi-grid synchronization in V100 platform. In each chart, the number of thread blocks per Stream Multiprocessor (Block Per SM) increases vertically, while the number of threads per thread block (Thread Per Block) increases horizontally.

5 GPU and 6 GPU.

Figure 3.13 shows that multi-grid synchronization outperforms the multi-device kernel launch function as an implicit barrier. On the other hand, as long as the program is not oversubscribed, i.e., no more than 1024 threads per SM, the performance of multi-grid synchronization is at most 3x slower than CPU-side barriers. Yet the difference is around 16 us, which is not an issue when using 8 GPUs. We argue that this minor cost should not discourage programmers from considering using multi-grid synchronization in their algorithms, given the utility provided in terms of simplicity of programming and avoiding reliance on third-party libraries such as openMP or MPI.

3.5. Considerations of Using CUDA Synchronization Instructions

In this section, we have identified several instances where synchronization instructions may not operate as expected.

3.5.1. Synchronization Inside a Warp

In this section, we examine synchronization at the warp level. To see if a barrier inside a warp is effective on all threads in the barrier, we run the code in Listing 3.4. In the ideal case, the timers

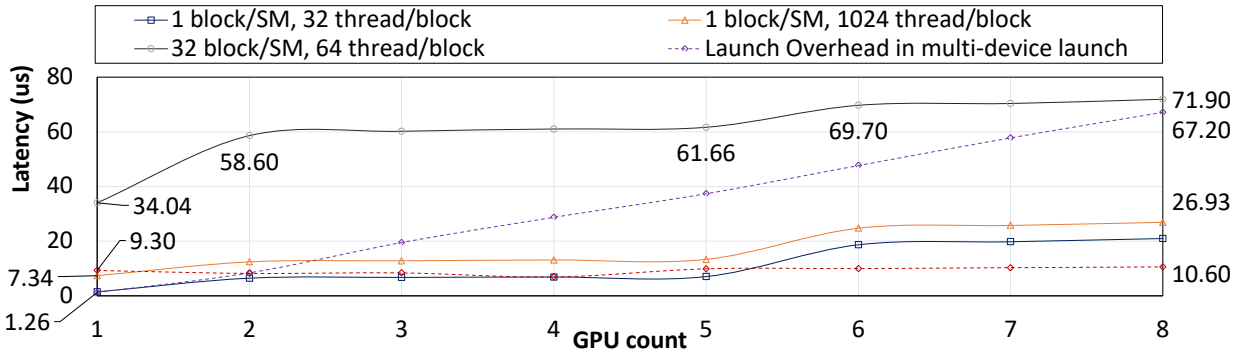


Figure 3.13: Comparison of implicit barriers performance: multi-device launch vs. CPU-side barriers and multi-grid synchronization across 8 GPUs in DGX-1

Listing 3.4: Comparison of implicit barriers performance: multi-device launch vs. CPU-side barriers and multi-grid synchronization across 8 GPUs in DGX-1

```
1 if(tid==0){timer(start);sync;timer(end);}
2 else if(tid==1){timer(start);sync;timer(end);}
3 ...
4 else if(tid==30){timer(start);sync;timer(end);}
5 else{timer(start);sync;timer(end);}
```

in all threads in the warp before the barrier are smaller than the timers after the sync in every thread. We test all the synchronization methods.

Results show that P100 does not assure all threads inside a warp are blocked at the barrier (also the shuffle operation does not work correctly in this code either), which we believe explains why the latency of warp level synchronization in P100 is as fast as Table 3.2 shows.

In V100, we observed the anticipated behavior (likely due to the fact that in V100 each thread has its own program counter). Figure 3.14 shows our observation when calling tile synchronization. We observed the same phenomenon when running all other synchronization instructions in both V100 and P100.

3.5.2. Deadlocks in Synchronizing Partial of Thread Groups

In this section, we examine the behavior of synchronization with a subset of a thread group: **Whether synchronizing a subset of a group cause a deadlock or not?**

We implement a test suite based on the codes in Listing 3.5, to see what happens when part of a thread group calls the synchronization function. We test through every granularity including threads, warps, blocks, and GPUs. As a result, we observed deadlocks when we synchronize parts of blocks in a grid group, a multi-grid group, and when we synchronize parts of GPUs in a multi-grid group. In summary, one should be careful, after initializing a grid group or a multi-grid group, since current CUDA does not support synchronizing sub-groups inside a grid group or a multi-grid group.

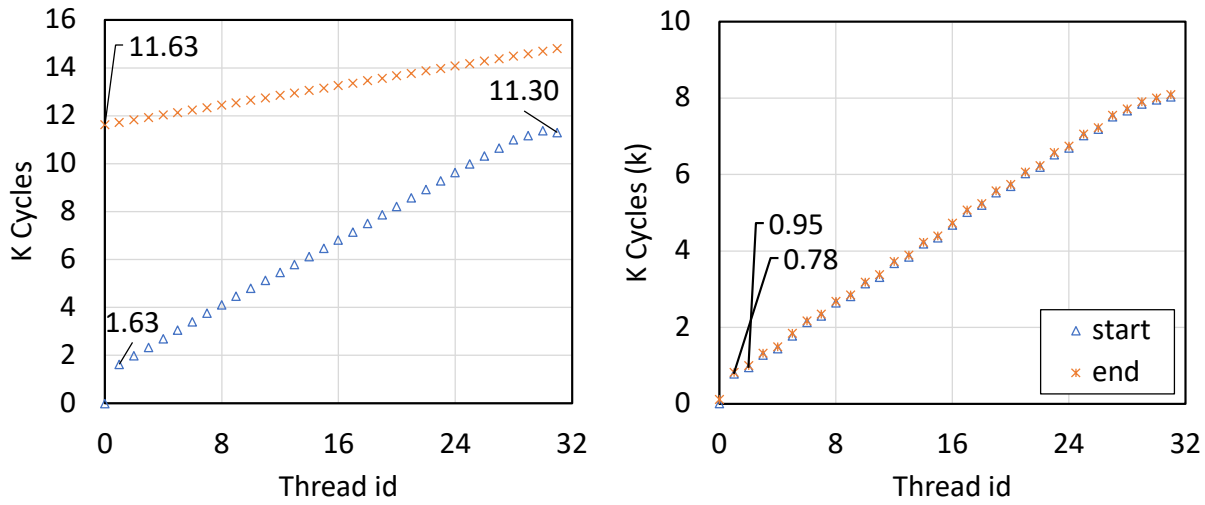


Figure 3.14: Timer of threads inside a warp when calling tile synchronization in V100 (left), and in P100 (right) in code sample of Listing 3.4

Listing 3.5: Code example to synchronize subgroup

```
...
//ensure that only part of a group calls the synchronization function
if(sync[unit_id]==1){sync;}
else if(sync[unit_id]==2){;}
...
```

3.6. CASE STUDY: Reduction Operator

To show how the knowledge of the performance features of synchronization methods can benefit code development, we use the reduction operator (summing the elements of an array) as a case study in this section. With its low operational intensity, the reduction operator constitutes a memory-bound kernel.

Harris et al. [86] did a notable work that focused on optimizing the reduction operator in CUDA. They studied several optimizations and optimized the reduction operator for maximum memory bandwidth utilization. Additionally, Luitjens et al. [87] introduced the shuffle primitive in reduction. The optimized reduction kernels can be found in CUDA SDK samples [12]. There are other similar optimization strategies [88, 89]. To the best of the authors' knowledge, all previous strategies didn't quantitatively compare synchronization methods in different implementations. In this section, we will demonstrate how to capitalize on the analysis in previous sections to make implementation decisions based on the input size and number of workers involved. This approach can be applied to optimize any of the previous reduction implementations and many other code generation frameworks [90].

In addition to using single GPU synchronization methods in optimizing input size, multi-grid

Table 3.3: Projected concurrency of the two configurations in two scenarios

scenario	**Configurations**	**Bandwidth B/cycle**		**Latency cycle**		**Concurrency B**	
		V100	P100	V100	P100	V100	P100
1	basic: 1 thread.	0.62	0.43	13.0	18.5	8	8
	more: 1 warp	19.6	13.8	13.0	18.5	256	256
2	basic:32 thread	19.6	13.8	13.0	18.5	256	256
	more:1024 thread	215	141	13.0	18.5	2796	2615

synchronization has a programmability benefit for multi-GPU systems. In dense systems, such as Nvidia DGX-1 and DGX-2, the peer access feature enables one GPU to access the memory of another GPU. In this case, multi-grid synchronization provides an easy way to ensure sequential consistency.

Another potential benefit that does not appear in this case is the potential to improve data reuse by replacing several kernel invocations with a single persistent kernel using multi-grid synchronization. An example would be replacing kernel invocations in iterative stencil methods with a persistent kernel that includes the time loop inside the kernel. The detailed discussion is in Chapter 4.

3.6.1. Performance Analysis

Overview

This performance model is based on the following assumptions:

- The throughput is indifferent to the size of the problem (for any problem size that fully utilizes the device).
- The cost of synchronization is the main cost of multi-threading. And it influences the performance model by increasing the Latency $\mathbb{L}$

As such we use the following formulation as a performance model to infer runtime T:

$$T = \max(\mathbb{L}, \frac{N}{\mathbb{THR}}) = \mathbb{L} + \max(0, \frac{N - \mathbb{C}}{\mathbb{THR}}) \tag{3.6}$$

Figure 3.15 visually explains Equation eqt:little.

We can use Equation 3.8 to decide when to use fewer parallelism settings. Migrating to more parallelism means longer Latency, including synchronization overhead, higher throughput, and a higher different concurrency to saturate the device. From this equation we can imagine three different scenarios:

1. If the input size is not larger than the concurrency of the "basic" situation, using fewer threads would always be more profitable because the longer latency makes more parallelism unprofitable.

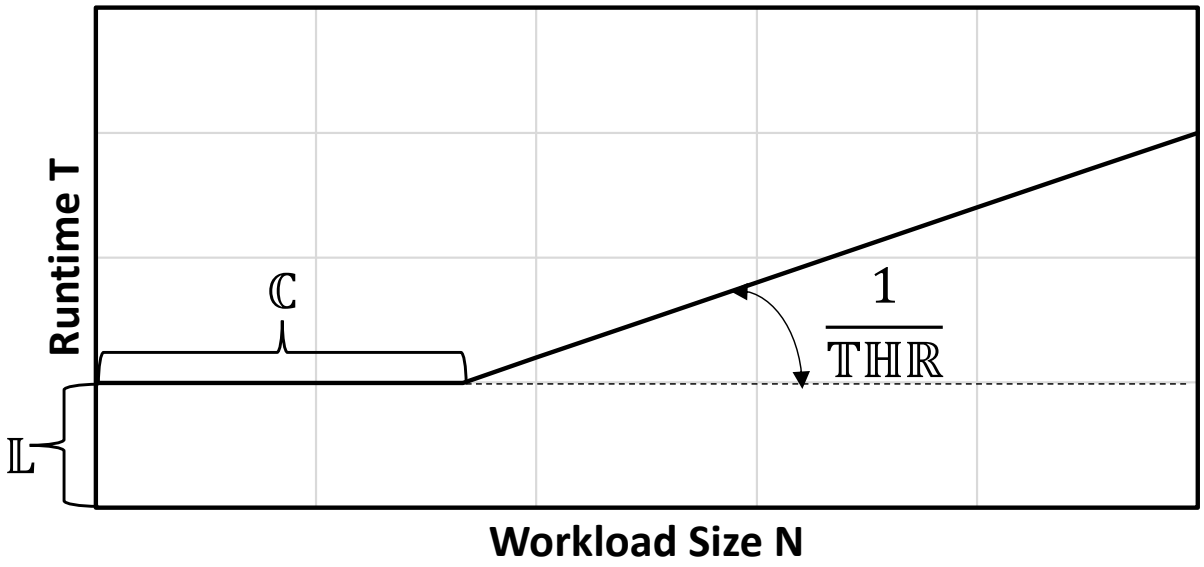


Figure 3.15: Visualize Equation 3.6

2. If the input size is larger than the concurrency of "basic" and no larger than the concurrency of "more" parallelism, we can use Equation 3.9 to compute the switching point.
3. If the input size is larger than the concurrency of "more" parallelism. We can use Equation 3.10 to know at which point we should migrate to using more parallelism.

$$\mathbb{L}^{'} = \mathbb{L}_0 + \mathbb{L}_{sync} \tag{3.7}$$

$$T_0 : T^{'} \rightarrow \mathbb{L}_0 + \frac{Max(0, N - \mathbb{C}_0)}{\mathbb{THR}_0} : \mathbb{L}^{'} + \frac{Max(0, N - \mathbb{C}^{'})}{\mathbb{THR}^{'}} \tag{3.8}$$

$$N_m < (\mathbb{L} + \mathbb{L}_{sync}) * \mathbb{THR}_0 \tag{3.9}$$

$$N_l > \frac{(\mathbb{L}_{sync}) * \mathbb{THR}^{'} * \mathbb{THR}_0}{\mathbb{THR}^{'} - \mathbb{THR}_0} \tag{3.10}$$

Microbenchmarking and Prediction

In the context of the GPUs examined in this paper, the bottleneck of the reduction algorithm becomes the device memory bandwidth when the input size is sufficiently large. Consequently, we use a memory bandwidth STREAM microbenchmark to emulate the performance of the reduction operation. To ensure this microbenchmark accurately represents reality, we incorporate 'add' instructions to mimic the computational workload in reduction operations.

We aim to determine when to use a single thread or a single warp barrier and when implementing a multi-GPU barrier would be more beneficial. Instead of exploring every possible case, we focus

Table 3.4: Predicting the switching point between two configurations in the two scenarios

Scenario		**Synchronize latency* cycle**		**Switch point B**	
		V100	P100	V100	P100
1	1 warp N_l	110	155	70	70
	1 warp N_m	-	-	76	75
2	1024 thread N_l	420	2135	9076	32681
	1024 thread N_m	-	-	8501	29737

∗: 5 times synchronization

Listing 3.6: Code example of reduction operator with explicit device synchronization

```
//works in both single and multi GPU
__global__ void ExplicitGPU(...){...
    while(step.notfinish()){
        //directly store data in the target GPU
        dest[step][threadid] = summing(src[step][threadid], rangeofthreadid);
        grid.sync();//explicit synchronize;
    }
    if(gpu_id==0){
        sum=block_reduce(src[0][0], ...);
        if(threadid==0){output[threadid]=sum;}
    }
}
```

on two configurations here, though these principles can be extrapolated to other scenarios:

- To use a single thread or single warp barrier
- To use a single block with 1024 threads or with 32 threads

Normally in the two configurations we mentioned, the data is kept in shared memory or cache, so we only measure shared memory in this Subsection. Table 3.3 shows the results of bandwidth (throughput), latency, and concurrency.

Take the double type as an example (8 Bytes). In this case, in both configurations, the input size exceeds the concurrency of both "basic" and "more" settings. Hence we only need to consider N_l in Equation 3.10. Table 3.4 shows the results.

Table 3.4 shows that it is better to compute 32 data points with a warp; second, there would be no benefit to compute 1024 data points with 1024 threads per block. Further experimental results validate these predictions.

It's worth noting that another potential overhead introduced by synchronization could be the clearance of the instruction pipeline. Threads might require additional time to saturate the pipeline again, implying that the actual switch point could be greater than what we've projected in Table 3.4.

3.6.2. Implementations

Detailed implementations are available at https://github.com/neozhang307/Reduction.

Listing 3.7: Sample kernel implementations for reduction operator with implicit device synchronization

```
__global__ void Kernel1(...){
    ...
    uint i = threadid + blockid * blockdim;
    sum = summing(src[step][i], rangeofi);
    output[i]=sum;
    ...
}
__global__ void Kernel2(...){
    ...
    sum=block_reduce(...);
    if(threadid==0){output[threadid]=sum;}
    ...
}
```

Table 3.5: Runtime (cycles) to sum 32 values (double precision)

	serial	nosync *	volatile & tile	tile	coa	tile shuffle	coa shuffle
V100	299	89	237	237	237	164	1261
P100	383	112	282	281	251	212	1423

**result of no synchronization version is incorrect*

Single GPU Reduction Operator

In this Subsection, we directly apply the knowledge in Section 3.6.1 in implementing device-wide reduction. Listing 3.6 shows the code of reduction with explicit synchronization and the upper part of Listing 3.8 shows the code of reduction operation with implicit synchronization for a single GPU.

Multi-GPU Reduction Operator

We use the code in Listing 3.6 and implicit Multi-GPU code in Listing 3.8.

We want to emphasize here the benefit of programming. We can easily implement explicit (Listing 3.6) kernel based on implicit barrier kernel code (Listing 3.7), i.e. a single persistent kernel with grid synchronization is necessary. And when extending it to multi-GPU, the complexity of managing several GPUs with CPU threads or processes is eliminated, such that the kernel function requires no knowledge of the hardware structure.

3.6.3. Evaluation

Table 3.6: Bandwidth (GB/s) of different reduction methods

	implicit	grid sync	CUB	CUDA sample	theory
V100	865.40	855.59	849.39	852.98	898.05
P100	592.40	590.85	543.96	590.65	732.16

Listing 3.8: Code example of reduction operator with implicit device synchronization in single GPU and multiple GPUs

```
void implicitSingleGPU(...){...
    Kernel1<<<...>>>(...);//implicit synchronization
    Kernel2<<<...>>>(...);//implicit synchronization
    ...
}
void implicitMultiGPU(){...
    #pragma omp
    for num_threads(gpucount){...
        cudaDeviceSet(tid);
        Kernel1<<<...>>>(...);
        //gather data to one GPU that would do the remaining computation.
        while(step.notfinish()){
            cudaDeviceSynchronize();
            #pragma omp barrier;
            //transfer data from current GPU to another GPU
            transferdata(src[step][tid],dst[step][tid]);
        }
        cudaDeviceSynchronize();
        #pragma omp barrier;
        if(tid==0)Kernel2<<<...>>>(...);
        ...
    }
}
```

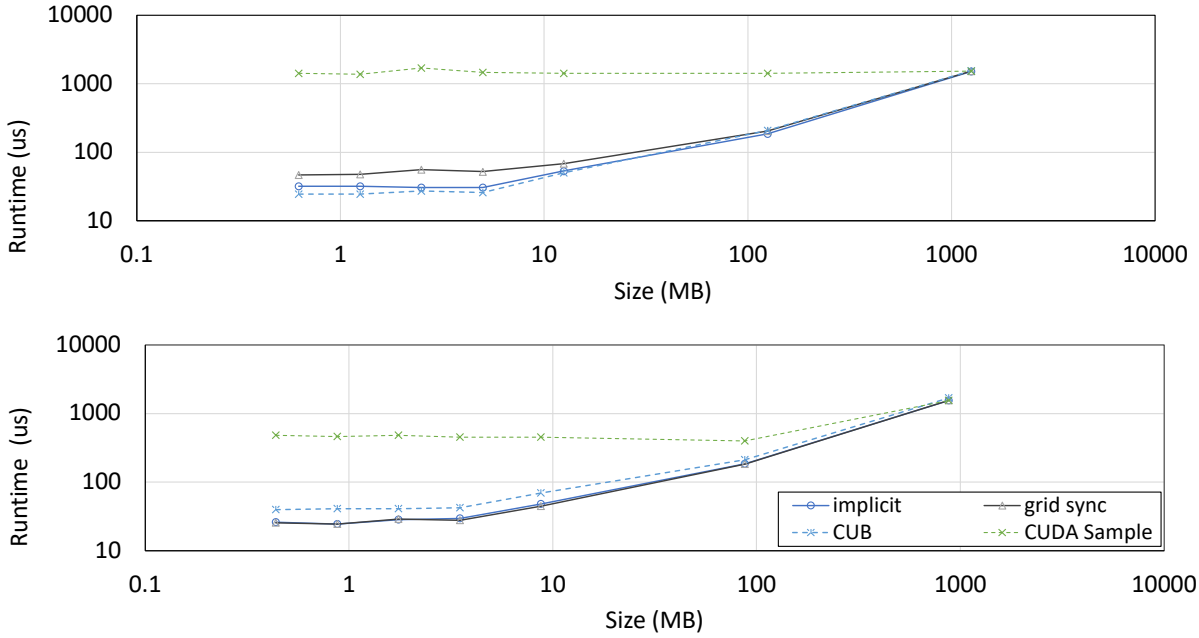


Figure 3.16: Performance comparison of reduction operation in single V100 (upper) and P100 (down)

Warp Level

We compare different warp level synchronization methods in the reduction kernel by observing their behavior in the current GPU platforms. Table 3.5 shows the result.

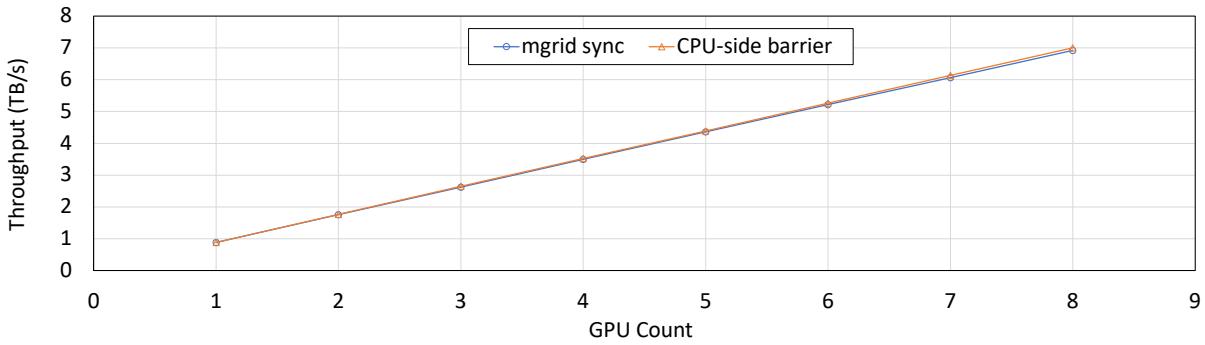


Figure 3.17: The throughput of reduce operator on DGX-1 across varying GPU amounts (weak scaling)

As shown in Table 3.5, when using the *volatile* qualifier for the input data, the performance of warp level synchronization is no worse than in the case without the volatile qualifier (shown as "tile" in the table). Accordingly, the warp level synchronization does not have much overhead other than to ensure memory consistency. We can conclude that warp level synchronization is no more than a memory fence in the current version of CUDA. We also observe that the results for using the shuffle operation with the tile group have the lowest latency.

Single GPU Level

The widely used GPU C++ library CUB [91] and CUDA SDK samples [12] include single GPU reduction implementations, and we compare the performance of those implementations with our implementation.

Table 3.6 shows the results. Our implementation is comparable to the state-of-the-art implementations on V100 and is noticeably better on P100.

Multi-GPU Level

Figure 3.17 shows the results of using implicit and explicit barriers for the reduction operator. Though hard to notice, an implicit barrier is always slightly better than the multi-grid synchronization method. As section 3.4.1 mentioned, the overhead associated with cooperative multi-launch might cause this performance discrepancy.

3.7. Conclusion & Take-out Notes

In this chapter, we delve into the evolution of Nvidia GPUs over the past decade and introduce microbenchmarks tailored for such investigations. We provide detailed microbenchmarks for synchronization methods in Nvidia GPUs. Our findings can be summarized as follows:

- Gap between compute and memory bandwidth is expending
- The capacity of on-chip resources has seen significant growth.

- It is easier to achieve higher theoretical performance with lower occupancy, implying that the device can be saturated more easily.
- Synchronization has overhead, though its overhead decreases over time, and its overhead is generally negligible in large data sets.

As such, we introduce two strategies:

- Combine time steps.
- Maintain minimal parallelism.

CHAPTER 4

PERKS: a Locality-Optimized Execution Model for Iterative Memory-bound GPU Applications

Iterative memory-bound solvers commonly occur in high-performance computing (HPC) codes.

Typical GPU implementations often encompass a host-side loop that invokes the GPU kernel as frequently as the required time or algorithmic steps. Each kernel's termination implicitly fulfills the barrier requirement after every time step progression in the solution. This chapter proposes an execution model for running memory-bound iterative GPU kernels: PERsistent KernelS (PERKS). This model moves the time loop inside a persistent kernel and leverages device-wide barriers for synchronization. We then reduce the traffic to device memory by caching a subset of the output in each time step in the unused registers and shared memory. PERKS can be generalized to any iterative solver: they largely depend on the solver's implementation. We elucidate the design principle of PERKS and demonstrate the effectiveness of PERKS for a wide range of iterative 2D/3D stencil benchmarks (geomean speedup of 2.12x for 2D stencils and 1.24x for 3D stencils over state-of-art libraries), and a Krylov subspace conjugate gradient solver (geomean speedup of 4.86x in smaller SpMV datasets from SuiteSparse and 1.43x in larger SpMV datasets over a state-of-art library). All PERKS-based implementations are available at https://github.com/neozhang307/PERKS.

The key contributions of this chapter include:

- We propose PERKS, an execution model that explicitly exploits the large volume of unused on-chip resources to reduce memory traffic in iterative memory-bound applications (Section 4.3).
- We provide analyses of the potential benefit of PERKS, and how to effectively port iterative solvers to PERKS (Section 4.4).
- We implement a wide range of iterative 2D/3D stencil benchmarks and a conjugate gradient solver as PERKS in CUDA. It is important to note that iterative stencils and Krylov subspace solvers are the backbones of numerous scientific and engineering codes. We include an elaborate discussion on implementation details and performance-limiting factors, such as problem sizes, concurrency, and resource contention (Section 4.5).
- We conduct a thorough evaluation (Section 4.6). PERKS-based implementation achieves geometric means speedups of 2.12x for 2D stencils and 1.24x for 3D stencils compared to several highly optimized state-of-the-art 2D/3D stencil libraries on A100 and V100. PERKS-based conjugate gradient achieves a geometric mean speedup of 2.48x compared to the highly GPU-optimized pro-

duction library Ginkgo [14] for SpMV datasets in SuiteSparse. For smaller datasets, the speedup goes up to 2.86x.

4.1. Introduction

GPUs are becoming increasingly prevalent in HPC systems. More than half the systems on the Top500 [50] list includes discrete GPUs, and seven of the systems in the top ten are GPU-accelerated (November 2022 list). As a result, extensive efforts goes into optimizing iterative methods for GPUs, for instance: iterative stencils [5, 8, 29, 30] used widely in numerical solvers for PDEs, iterative stationary methods for solving systems of linear equations (ex: Jacobi [33, 92], Gauss–Seidel method [31–33]), iterative Krylov subspace methods for solving systems of linear equations (ex: conjugate gradient [14, 93], BiCG [14, 94], and GMRES [14, 95]).

Although the device memory bandwidth of GPUs has been increasing from generation to generation, the gap between computing and memory is widening. Given that iterative stencils and implicit solvers typically have low arithmetic intensity [5], significant efforts go into optimizing them for data locality. These included moving the bottleneck from device memory to on-chip scratchpad memory [6] or cache [48], or further pushing the bottleneck to the register files [1, 8]. Those efforts become increasingly effective since the aggregate volume of register files and scratchpad memory capacity are increasing with newer generations of GPUs [96]. In iterative solvers, due to spatial dependencies, a barrier is typically required at the end of each time step (or several time steps when doing temporal blocking [5]). That is to ensure that advancing the solution in time step k would only start after all threads finish advancing the solution in time step $k-1$. Invoking the kernels from the host side in each time step acts as an implicit barrier, where the kernel invocation in time step k would happen after all threads of the kernel invocation at time step $k-1$ have finished execution. In-between kernel invocations, data stored in registers and scratchpad memory would be wiped out, and the next kernel invocation would start by reading its input from the device memory.

One opportunity to improve the data locality is to extend the lifetime of the kernel across the time steps and take advantage of the large volume of register files and scratchpad memory to reduce traffic to the device memory in between time steps. This paper proposes a generic model for running iterative solvers on GPUs to improve data locality. PERsistent KernelS (PERKS)[1] have the time loop inside them, instead of the host, and use the recently supported device-wide barriers (in CUDA) for synchronization. Next, we identify the cache-able data in the solver: the data that is the output of time step $k-1$ and input to time step k, as well as the repeatedly loaded constant data. Finally, we use either the scratchpad memory or registers (or both) to cache the data and reduce the traffic to device memory.

The basic concept and implementation of PERKS are relatively simple, which we argue is essen-

[1] In this dissertation, we use PERKS, interchangeably, to refer to our proposed model and, as an abbreviation of PERsistent KernelS

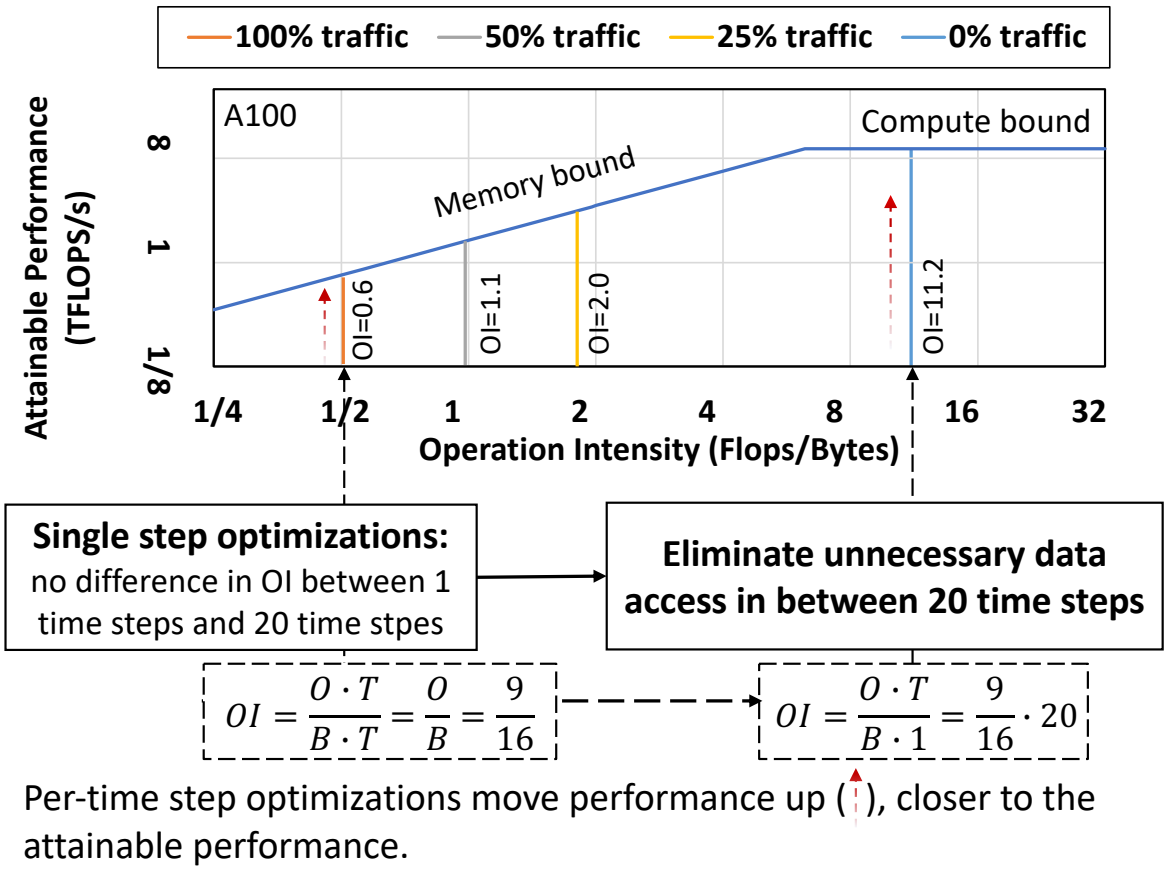


Figure 4.1: The roofline model of a 2D 5-point Jacobian stencil kernel (dp), running $T = 20$ time steps, with a domain size of 3072^2 on A100 GPU. Per-time step optimizations only improve the iterative stencil kernel to get closer to the attainable performance. Reducing memory traffic between time steps can increase the attainable performance by increasing the aggregate operation intensity (OI) over all time steps. We also plot different operational intensities for a version of PERKS that reduces the data traffic in-between 20 time steps to 50%, 25%, and 0%.

tial for encouraging scientists and engineers to adopt PERKS in their iterative solvers implemented for GPUs, and other architectures. That being said, a challenging aspect we address in this paper is a detailed analysis of how and why PERKS is effective. The analysis requires an understanding of the effect of concurrency on performance. More particularly, to gain a deep understanding of why PERKS are practical and the limitations of architectural features, we study the effect of pressure on resources (particularly registers and shared memory). On top of that, we examine the effect of reducing the device occupancy while maintaining high enough concurrency to saturate the device.

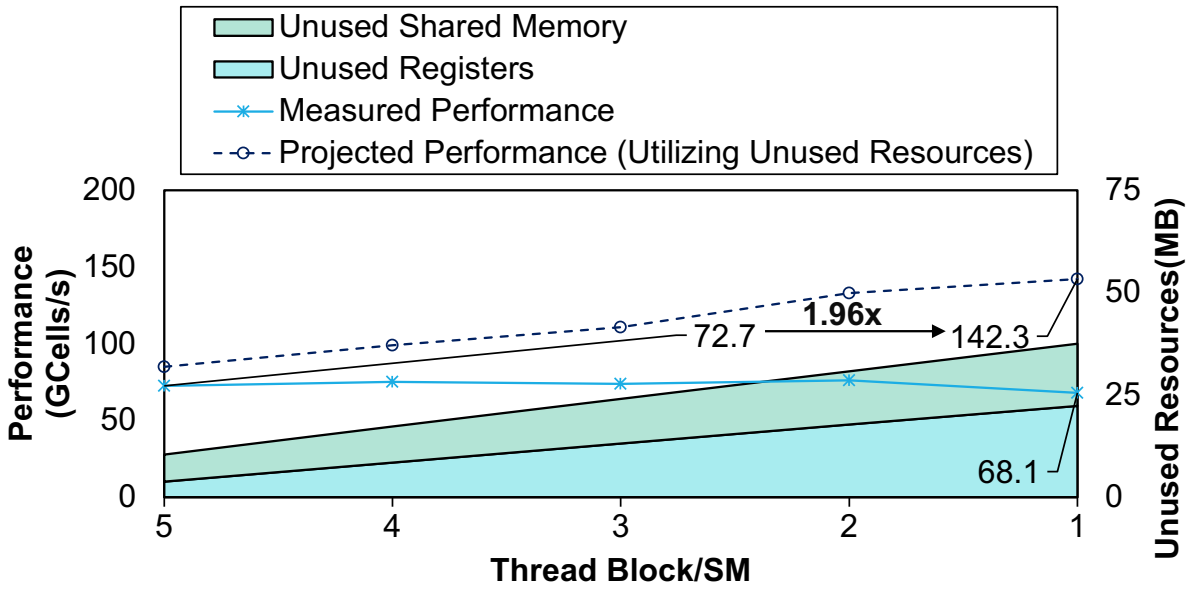


Figure 4.2: Performance of a double precision 2D 5-point Jacobian stencil kernel (3072^2) for different Thread Blocks per Streaming Multiprocessor (TB/SM) on A100. Filled regions indicate unused resources. Using one TB/SM and using all unused resources for caching can theoretically provide 1.96x speedup in this situation.

4.2. Motivation

4.2.1. Motivational Example

We use a motivational example of a double precision 2D 5-point Jacobian (2d5pt) stencil to motivate implementing iterative solvers as PERKS (Readers can refer to Equation 2.11 and Listing 4.1 for details of the 2d5pt stencil). ❶ Why PERKS: Optimizations for iterative methods focus on a single step to speed up iterative solvers. Single-step optimizations move the performance of the kernel closer to the highest possible attainable performance on the roofline model, yet will not influence the operational intensity. As Figure 4.1 shows, the optimizations used for the 2D 5-point stencil move the performance vertically at the same operational intensity value of the kernel. Temporal blocking schemes can horizontally move the operational intensity to the right side of the roofline. Yet resolving the neighborhood dependencies introduces redundancy [53, 72, 73] or hard-to-parallel complex geometrical tile shapes [54–56], and can cause register pressure [5]. In PERKS, we reduce the unnecessary data access between time steps. The target data traffic to reduce is in-between time steps (i.e., outside the solver) and hence is not subject to the neighborhood dependency issue in temporal blocking schemes. Figure 4.1 demonstrates how this idea works for a real stencil benchmark running on an A100 GPU for 20 time steps. By caching more of the domain in-between time steps, the operational intensity moves more to the right side of the roofline to be compute-

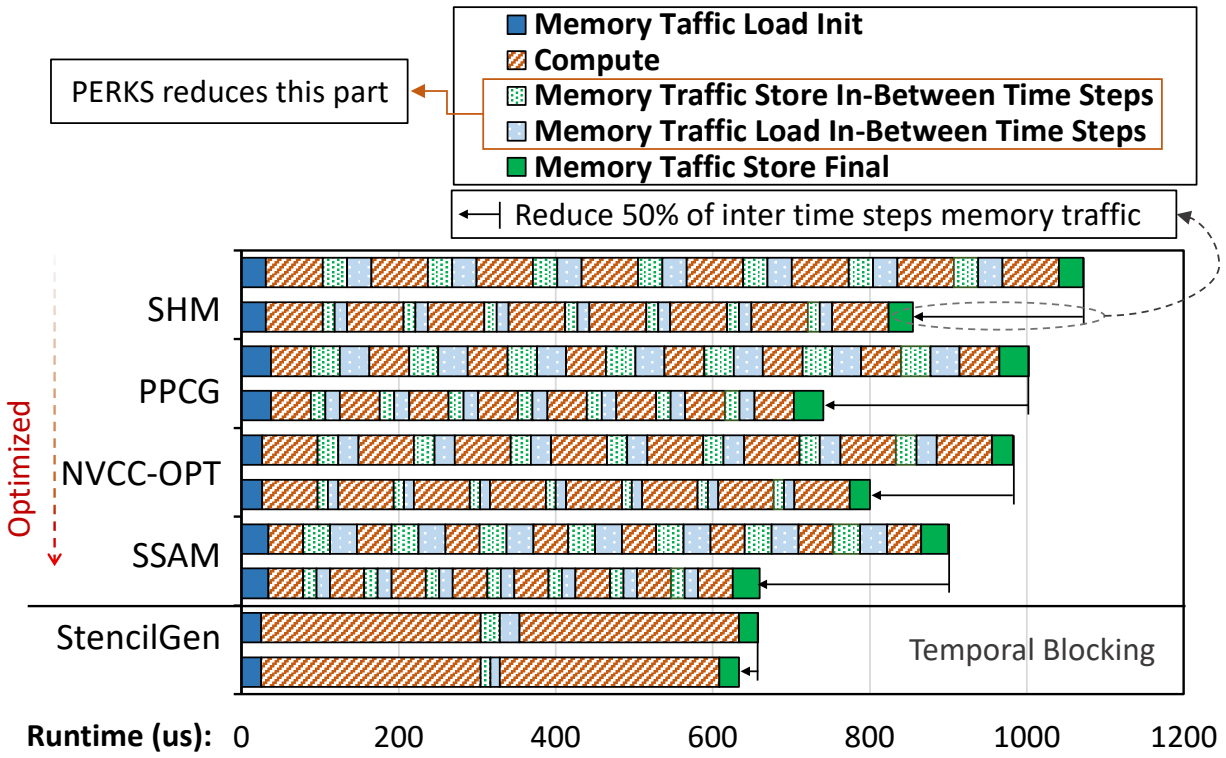


Figure 4.3: Runtime (8 time steps) of double precision 2D 5-point Jacobian stencil (3072^2) with different state-of-the-art optimization A100 GPU. PERKS aims to reduce the traffic to/from device memory in-between time steps. SHM [6] uses shared memory. PPCG is a code auto generation tool [7]. NVCC-OPT relies on auto optimization provided by the latest NVCC compiler version. SSAM [8] uses register to improve locality [9–11]. Stencilgen [4] applies temporal blocking. In the figure we also plot the total runtime of each implementation running 8 time steps, assuming we reduce 50% of the inter time step memory traffic. The results show that the more optimized (i.e. fewer proportion is spent in compute), the more performance improvement we expect from caching.

bound. This also demonstrates how PERKS is orthogonal to the per-time step optimizations; PERKS would improve the performance (by moving horizontally on the roofline) regardless of how optimized the baseline algorithm is at its operational intensity. ❷ The prospect of PERKS: Latency across all operations/instructions in newer generation GPUs has been significantly dropping [97]. As a result, often fewer numbers of warps are enough for CUDA runtime to hide the latency effectively and hence maintain high performance at low occupancy [65]. In Figure 4.2, we vary the number of thread blocks per streaming multiprocessor (TB/SM) and plot its performance (left Y-axis). For each TB/SM configuration, we plot the unused resources (shared memory and registers) on the right Y-axis. As the figure shows, **even when TB/SM = 5, more than 10.46 MB of shared memory and register files are not in use.** When TB/SM decreases, the performance is slightly fluctuated

(72.74 → 68.12 GCells/s [2]) while the freed shared memory and registers gradually increase. By reducing TB/SM to its minimum while maintaining enough concurrency to sustain the performance level, the projection from performance gain when caching a subset of the results in unused resources can improve performance by more than 1.96x. Figure 4.3 uses an alternative perspective, assuming that memory accessing and compute within a solver can not be overlapped. We profiled different 2d5pt state-of-the-art implementations. As Figure 4.3 shows, the compute part decreases the more optimized the stencil implementation is. The prospect of PERKS is to reduce this data movement time that dominates the runtime in highly optimized stencil implementations. Note that while temporal-blocking schemes do also reduce the data movement to some extent, they cannot be generalized to all iterative solvers. Additionally, resolving temporal and spatial dependency adds compute overhead and can lead to increased register pressure that limits the degree of temporal blocking on GPUs [5].

4.3. PERKS: Persistent Kernels to Improve Locality

PERsistent Kernels (PERKS) is a generic execution model for running iterative solvers on GPUs to improve data locality by taking advantage of the large capacity of register files and shared memory. As Figure 4.4 illustrates, in PERKS, we move the time stepping loop from the host to the device, and use CUDA's grid synchronization API as a device-wide barrier at each time step. This way, we expose the temporal data locality across time steps to thread blocks. We then use the register files and shared memory to reduce traffic to the global memory by caching the domain (or a subset of it) in-between time steps.

4.3.1. Assumptions and Limitations

The techniques discussed in this paper are based on the following assumptions about the applications.

Target Applications: In this paper, we target iterative kernels that are bounded by memory bandwidth. Although execution in a PERKS fashion makes no assumptions on the underlying implementation, optimal PERKS performance can sometimes require minor adaptations to the kernel. The changes, for instance, can be as simple as changing the thread block and grid sizes to reduce over-subscription or more elaborate as favoring a specific SpMV method from the space of SpMV methods in the case of the conjugate gradient (more details in Section 4.5.3). Finally, despite not reporting results for compute-bound iterative kernels, it is important to note that compute-bound iterative kernels could potentially also benefit from becoming PERKS if the kernel generates memory traffic in-between iterations that CUDA runtime can not effectively overlap with computation.

PERKS in Distributed Computing: PERKS in this paper is demonstrated on a single GPU. In distributed applications that require halo regions (e.g., stencils), PERKS can potentially

[2] GCells/s denotes giga-cells updated per second.

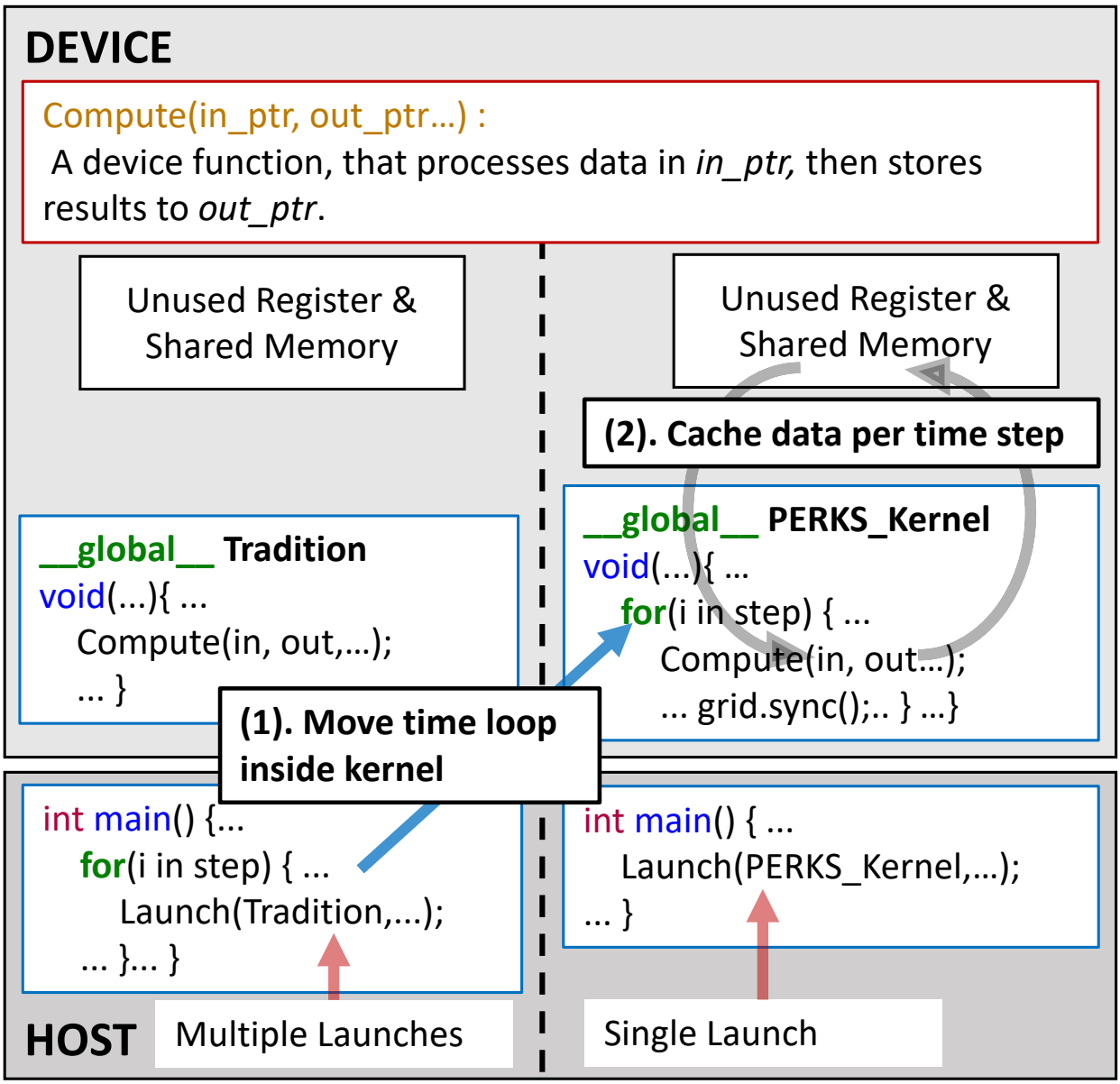


Figure 4.4: Changing a traditional iterative CUDA kernel (time loop on the host) to PERKS: 1) move the time loop from the host code to the kernel code and use grid synchronization between time steps. 2) Cache data between time loops on the unused shared memory and register files. The compute portion of the kernels does not notably change, i.e., no need to change the original algorithm when using PERKS.

be used on top of communication/computation overlapping schemes [98,99]. In overlapping schemes, the boundary points computed in a separate kernel would not be cached, while the kernel of the interior points would run as PERKS to cache the data of the interior points. PERKS could also be used with communication-avoiding algorithms (e.g., communication-avoiding Krylov methods [100])

Practicality of PERKS: A wide range of iterative solvers (particularly iterative stencils) can be written as PERKS. However, it should be mentioned that there are applications in which the time stepping loop (on the host) comprises different GPU kernels. For instance, in production libraries, conjugate gradient (and Krylov solvers in general) are typically implemented as different kernels corresponding to different steps in the algorithm. In such case, the kernels have to be fused [101,102] first before transforming them to PERKS.

Use of Registers: PERKS uses registers and shared memory for caching data in-between time steps. It should be noted that there are no guarantees that the compiler releases all the registers after the compute portion in each iteration is finished (with Nvidia's nvcc compiler we did not observe such inefficiency). If such register reuse inefficiency exists, imperfect register reuse by the compiler could result in fewer registers being available for caching and leaves only shared memory to be used for caching. PERKS would not be effective if the target kernel consumes all on-chip resources (both register file and shared memory) even in its minimal occupancy.

Iterative Solvers as PERKS: While this paper's focus is to demonstrate the PERKS model for iterative stencils and Krylov subspace methods (conjugate gradient), the discussion in this section (and paper in general) is applicable to a high degree for other types of iterative solvers. That is since PERKS is not much concerned with the implementation of the solver and only loads/stores the domain (or a subset of it) before/after the solver part in the kernel, under resource constraints. Iterative solvers that use the same flow expressed in Figure 4.4 can, in principle, be ported to PERKS (with relative ease). Generally speaking, the porting process is as follows: move the time step outside the kernel to be inside the kernel, add grid synchronization to ensure dependency, and store/load a portion of the input or output to cache: either shared memory and/or register (using register arrays). More details on porting kernels to PERKS are in Section 4.5.1.

4.3.2. Cache Policy: What to Cache and Where to Cache

A caching policy is required to determine which portion of the domain (or data) to cache. When the entire domain (or data) can fit in register files and shared memory used for caching, the entire algorithm can run from the cache (this is particularly useful in the cases of strong scaling where per node domain size becomes smaller as the number of nodes grows). When only a fraction of the domain can be cached, which is more common, a policy must select the data to prioritize for caching. In the following sections, we elaborate on the caching policy.

Considerations for dependency between threads Blocks

Algorithms that do not require dependency between the thread blocks can use the cache space most efficiently because all the load and store transactions to global memory can be eliminated. The dependencies within the thread block are resolved by using either the shared memory or shuffle operations.

In iterative solvers, there is often neighbor dependency (ex: a stencil kernel where a cell update relies on values computed in neighboring threads). In such a case, caching the results of the threads at the interior of the CUDA thread blocks eliminates the stores and loads from global memory. However, the threads at the boundary of thread blocks would continue to store and load from global memory (since the shared memory scope is the thread block). When the capacity of the cache is large enough, w.r.t. the domain size to be cached, the performance drawback would be negligible.

Identifying which data gets priority to be cached

In many cases, the capacity of register files and shared memory is limited, i.e., it is impossible to cache the entire domain/input. In cases where all the domain/input array elements are accessed at the same frequency, one could assume it is unnecessary to use a cache policy that prioritizes specific parts of the domain/input. However, this is not always true.

Take iterative stencil as an example. The data managed by the threads at the boundary of the thread block is stored in the main memory to be accessed by the neighboring threads blocks in the following iterations; caching those boundary elements saves one load operation. On the other hand, data at the interior of the thread block is not involved in inter thread block dependency; caching saves one load and one store operation. Finally, data in the halo region is updated at each time step; there is no benefit in caching the layers in the halo region. To conclude, the priority in caching yielding the highest reuse would be: $Data_{no\_inter\_TB\_dependency} > Data_{inter\_TB\_dependency}$, i.e., the priority is to cache the data of the interior threads of the thread block, followed by the data of the threads at the boundary of the thread block, and no caching for the halo region.

There could be situations that different data arrays could be potentially cached for other iterative solvers, such as conjugate gradient, unlike a single domain array in stencils. The cacheable variables and usage per array element for the conjugate gradient solver are as follows: a) one load and no stores for the matrix $\mathbf{A}$, and b) three loads and one store for the residual vector $\mathbf{r}$. So by assuming that each operation accesses data in a coalesced access pattern, it would be more effective to cache vector $\mathbf{r}$. As a result, the ideal cache priority is $\mathbf{r} > \mathbf{A}$.

To summarize, while PERKS does not touch on the compute part of the original kernel, attention should be given to identify the ideal caching policy for each solver implemented as PERKS. That being said, one could assume this step can be automated by using a dedicated profile-guided utility (or even sampling from the profiler directly) to aid the user in swiftly identifying an ideal caching policy based on the access patterns and frequency of access of data arrays in the solver.

4.3.3. Caching Schemes

This section identifies three different caching schemes that could be used in PERKS. Figure 4.5 shows the details.

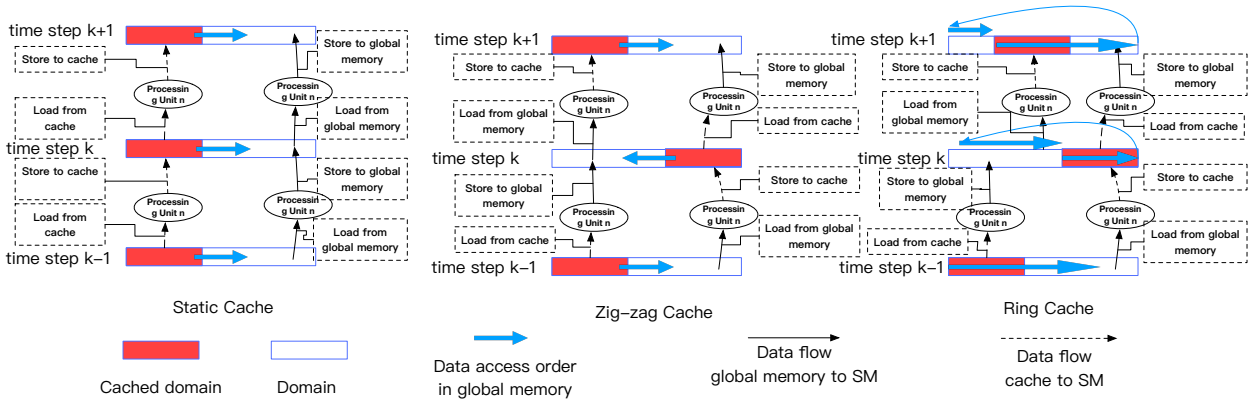


Figure 4.5: Different caching schemes with PERKS. *Static cache:* computed portion of the domain is cached, statically. *Zig-zag cache:* the direction of domain traversal is zig-zagged, where the latest computed portion of the domain in each iteration is the first to compute in the following iteration. *Ring cache:* computing in each iteration continues from the point in the domain that was cached and upon reaching the end of the domain we continue from the beginning of the domain in a ring-fashion.

- **Static cache**. The static cache is the most straightforward scheme. Before starting the kernel, we statically map an address range to the cache. When processing this address range, PERKS would directly access data from the cache.
- **Zig-zag cache**. The Zig-zag Cache scheme reverses the execution order in each iteration. We store the data to cache at the end of the time step k in the shared memory and registers. Next, we load the cached data at the beginning of the time step $k+1$. The advantage of this scheme is that it might get some added performance benefit from the L2 cache.
- **Ring cache**. Ring cache is an alternative to Zig-zag at which the computation in step $k+1$ continues from where it stopped in step k and one continues to the beginning of the domain when the end of the domain is reached (i.e., a ring-fashion). The ring cache preserves the same advantage that the zig-zag cache scheme has while avoiding the complexity of reversing the domain traversing order in each step.

Static cache is easy to implement. This scheme applies to any iterative method having repeated uniform accesses to isolable data regions. Both iterative stencils and Krylov subspace methods can use this caching scheme. **Zig-zag cache** and **ring cache** are similar. However, they only apply to kernels with iterative access to a single continuous domain (e.g., iterative stencils). These schemes have higher code complexity yet can benefit from a higher hit rate in the L2 cache. We tested all three schemes for iterative stencils. On the GPUs we tested (Nvidia Volta V100 and Ampere A100), the zig-zag and ring cache scheme show nearly no performance benefit over the static cache scheme. Profiling indicates that the performance gain from additional L2 cache hits is outweighed by the performance decrease caused by register pressure resulting from the additional complexity of altering the domain traversal directions in every iteration. Considering the trend of

high-capacity SRAM (and in GPUs in particular [103]), the zig-zag and ring cache schemes could benefit future GPU designs with large capacity L2 caches.

4.4. Performance Analysis

This section introduces a performance model that serves the following purposes. First, we propose a projection of achievable performance that we compare with measured results to detect abnormal behavior or implementation shortcomings. We relied on this projection to analyze our PERKS implementation quality (Section 4.4.2). Second, we identify the bounds on reducing concurrency before performance regression and use concurrency to explain potential optimizations for further performance improvement (Section 4.4.4). It is worth mentioning that concurrency analysis is not required for porting kernels to PERKS; we use the analysis to understand the feasibility of PERKS in practice and address its implication on performance.

4.4.1. Overview

This performance model relies on three performance attributes: a) measured performance $\mathbb{M}$ of our PERKS implementation, b) the projected peak performance $\mathbb{P}$ achievable on a given GPU, and c) the efficiency function $\mathfrak{E}()$ describing the efficiency of the given kernel running on the device. More specifically, $\mathfrak{E}()$ is a function of the concurrency exposed by the software $\mathbb{C}_{sw}$ and the concurrency required by the hardware $\mathbb{C}_{hw}$. The relation of measured performance to projected peak performance becomes:

$$\mathbb{M} = \mathbb{P} \times \mathfrak{E}(\mathbb{C}_{sw}, \mathbb{C}_{hw}) \tag{4.1}$$

We discuss projected peak performance in the following section. A detailed discussion of the efficiency and concurrency functions is in Section 4.4.3.

4.4.2. Projecting Peak Achievable Performance

We rely on the figure of merit as the performance metric in this analysis. In stencils, we use the giga-cells updated per second (GCells/s) [5, 8]. Given the conjugate gradient solver's memory-bound nature, we use sustained memory bandwidth as a metric, following other works on conjugate gradient [14]. Due to space limitations, this section mainly focuses on stencils to explain the performance analysis. Without loss of generality, the analysis applies to other cases (ex: conjugate gradient) by adjusting the performance metric and code concurrency accordingly.

We use a simple performance model inspired by the roofline model [27, 28]. The model's utility is to project the upper bound on performance based on reducing global memory traffic. This model, in turn, helps us in this paper to identify performance gaps in our PERKS implementation and later inspect the reasons for those gaps.

In a kernel implemented as PERKS, the bottleneck could either be the global memory bandwidth or the shared memory bandwidth (if the PERKS caching scheme moves the bottleneck to become the shared memory bandwidth). We don't assume the registers to be a bottleneck since we

assume that we avoid register pressure as long as we ensure that no register spilling occurs.

We assume a total domain of size $\mathbb{D}$ bytes, the cached portion to be $\mathbb{D}_{cache}$ bytes, and the uncached portion to be $\mathbb{D}_{uncache} = \mathbb{D} - \mathbb{D}_{cache}$ bytes. The cached portion of the domain data would be divided between registers and shared memory (since we cache in both registers and shared memory): $\mathbb{D}_{cache} = \mathbb{D}^{sm}_{cache} + \mathbb{D}^{reg}_{cache}$. For N time steps, assuming the number of bytes stored to global memory in each time step is $\mathbb{S}_{gm}$ and the number of bytes loaded is $\mathbb{L}_{gm}$, the total global memory bytes accessed $\mathbb{A}_{gm}$ becomes:

$$\mathbb{A}_{gm}(\mathbb{D}) = N \cdot (\mathbb{L}_{gm} + \mathbb{S}_{gm}) = 2 \cdot N \cdot \mathbb{D}_{uncache} + 2 \cdot \mathbb{D}_{cache} \tag{4.2}$$

When the kernel is bounded by global memory bandwidth, i.e., the volume of cached data does not move the bottleneck from global memory to shared memory, for the global memory bandwidth of $\mathbb{B}_{gm}$ and data type size of $\mathfrak{S}(type)$, the time $\mathbb{T}_{gm}(\mathbb{D})$ for accessing the global memory becomes:

$$\mathbb{T}_{gm}(\mathbb{D}) = \mathbb{A}_{gm}(\mathbb{D}) \cdot \mathfrak{S}(type)/\mathbb{B}_{gm} \tag{4.3}$$

In the case when the kernel is bounded by shared memory bandwidth, i.e., the volume of data cached in shared memory moves the bottleneck to be the shared memory bandwidth, the total shared memory (in bytes) accessed $\mathbb{A}_{sm}$ becomes:

$$\mathbb{A}_{sm}(\mathbb{D}^{sm}_{cache}) = N \cdot (\mathbb{L}_{sm} + \mathbb{S}_{sm}) = 2 \cdot (N-1) \cdot \mathbb{D}^{sm}_{cache} \tag{4.4}$$

Assuming $\mathbb{A}_{sm}(\mathbb{KERNEL})$ to be the shared memory originally used by the kernel, e.g., shared memory used in the baseline implementation of a stencil kernel to improve the locality, and $\mathbb{B}_{sm}$ to be the shared memory bandwidth, the time $\mathbb{T}_{sm}(\mathbb{D})$ for accessing the shared memory becomes:

$$\mathbb{T}_{sm}(\mathbb{D}^{sm}_{cache}) = \{(\mathbb{A}_{sm}(\mathbb{D}^{sm}_{cache}) + \mathbb{A}_{sm}(\mathbb{KERNEL})) \cdot \mathfrak{S}(type)\}/\mathbb{B}_{sm} \tag{4.5}$$

The projected best-case total time required for the PERKS kernel may be written as:

$$\mathbb{T}_{PERKS} = \max(\mathbb{T}_{gm}(\mathbb{D}), \mathbb{T}_{sm}(\mathbb{D}^{sm}_{cache})) \tag{4.6}$$

Accordingly, the projected peak performance ($\mathbb{P}$ in Equation 4.1) for the N time steps can be expressed as:

$$\mathbb{P} = \mathbb{D} \cdot N/\mathbb{T}_{PERKS} \tag{4.7}$$

We give an example of computing $N = 1000$ time-steps of a double precision 2D 5-point Jacobi stencil on A100. We use the domain size $\mathbb{D} = 3072^2$; the total cache-able region is $\mathbb{D}_{cache} = 3072 \cdot 1512$ leading to $\mathbb{T}_{gm}(\mathbb{D}) = 45.36$ ms. The total number of bytes for the halo accesses is

$\mathbb{A}(\mathfrak{H}(\mathbb{D}_{cache})) = 1000 \cdot 2 \cdot 108 \cdot (168 \cdot 2 + 256 \cdot 2)$. Thus $\mathbb{T}_{gm}(\mathfrak{H}(\mathbb{D}_{cache})) = 0.94$ ms. So $\mathbb{P}_{PERKS} = 3072^2 \cdot 1000/\mathbb{T}_{PERKS} = 187.62$ GCells/s. The measured performance is 112.09 GCells/s.

4.4.3. Concurrency and Micro-benchmarks

Reducing device occupancy increases resource availability for caching in PERKS (as illustrated earlier in Figure 4.2). On the contrary, reducing occupancy can lead to lower device utilization. To effectively implement PERKS, one has to reduce the occupancy as much as possible without sacrificing performance. Inspired by the findings of Volkov [65], we assume that the efficiency function $\mathfrak{E}$ reaches its peak point when the code provides enough concurrency to saturate the device (irrespective of the occupancy):

$$\mathfrak{E}(\mathbb{C}_{sw}, \mathbb{C}_{hw}) = 100\%, \text{if } \forall \mathbb{C}_{sw} \geq \mathbb{C}_{hw} \tag{4.8}$$

Where $\mathbb{C}_{sw}(\mathbb{OP})$ is the minimum number of concurrently executable instructions of the operation $\mathbb{OP}$ exposed by the launched kernel ($\mathbb{PAR}$ in Section 2.3), and $\mathbb{C}_{hw}(\mathbb{OP})$ is the maximum numbers of instructions of the operation $\mathbb{OP}$ that the device is capable of handling concurrently. Because this paper mainly focuses on memory-bound applications, the $\mathbb{OP}$ referred to in this paper are limited to data access operations, i.e. global memory load/store $\mathbb{C}(\mathbb{GM})$, shared memory load/store $\mathbb{C}(\mathbb{SM})$, and L2 cache load/store $\mathbb{C}(\mathbb{L}2)$.

Measuring $\mathbb{C}_{sw}^{SM}$

$\mathbb{C}_{sw}^{SM}(\mathbb{OP})$, the kernel concurrency at the Streaming Multi-processor (SM) level, can be computed based on the concurrency exposed by the threads of a thread block $\mathbb{C}_{sw}^{TB}(\mathbb{OP})$ and number of concurrently running thread blocks per SM TB/SM: $\mathbb{C}_{sw}^{SM}(\mathbb{OP}) = \mathbb{C}_{sw}^{TB}(\mathbb{OP}) \cdot TB/SM$.

Measuring $\mathbb{C}_{hw}$

According to Little's Law [104], the hardware concurrency $\mathbb{C}_{hw}$ can be determined by the throughput $\mathbb{THR}$ and latency $\mathbb{L}$ [65]:

$$\mathbb{C}_{hw}(\mathbb{OP}) = \mathbb{THR}(\mathbb{OP}) \cdot \mathbb{L}(\mathbb{OP}) \tag{4.9}$$

The throughput $\mathbb{THR}$ for data access operations are available in CUDA documentation [20,64]. We measure the latency $\mathbb{L}$ with commonly used microbenchmarks [81,105,106].

4.4.4. Concurrency Analysis

In this section, we briefly describe how we analyze the concurrency to reduce the occupancy of the original kernel to release resources for caching while sustaining performance. We conduct a static analysis to extract the data movement operations in the kernel. Note that we account for any barriers in the original kernels that could impact the concurrency of operations, i.e., we do not combine operators/instructions from before and after the barrier when we count the operators. Finally, we apply a simple model (Equation 4.8) to identify the least occupancy we could drop before the concurrency starts to drop. We focused on single precision here because we did not observe a

Table 4.1: Concurrency analysis of global memory accesses of a single precision 2D-5point Jacobi stencil kernel running on A100 (1000 time-steps on 3072^2 domain)

$\frac{TB}{SM}$	Used Reg. /SM	Unused Reg. /SM	GM Load op/SM	GM Store op/SM	Measured GCells/s
1	32KB	224KB	2580	2048	94.75
2	64KB	192KB	5160	4096	133.24
8	256KB	0KB	20640	16384	138.29

similar performance drop in double precision. The results summarized in Table 4.1 show that for a 2D 5-point Jacobi stencil kernel, we could reduce the original occupancy to $1/4^{th}$ while maintaining performance.

To understand the gap between the performances at 1 vs. 8 TB/SM ($94.75/138.29 = 68.52\%$), we inspect the efficiency function $\mathfrak{E}()$. The number of concurrent global and shared memory accesses in the 2D 5-point Jacobi stencil kernel is enough to saturate A100 when TB/SM=1. Accordingly, we get $\mathfrak{E}_(\mathbb{C}_{sw=j2d5pt,TB/SM\geq 1}, \mathbb{C}_{hw=A100}) = 1$, which would indicate that the observed gap in performance is not due to a drop in concurrency we did not model. While this confirms the effectiveness of the concurrency analysis (i.e., since the concurrency analysis resonates with the empirical measurements in Table 4.1), it does not uncover the source of the performance gap. Investigative profiling revealed that the concurrency for accesses in the L2 cache, not global memory, is impacted by reducing occupancy on A100 specific to the level that affects performance notably. More particularly, access to global memory for the halo region garners a high L2 cache hit rate. This effectively means that higher concurrency is necessary to saturate the L2 cache when hit rates are high. To confirm, we manually doubled the concurrency $\mathbb{C}_{sw}^{TB}$: the performance increased to 123.94 GCells/s with TB/SM=1 (from 68.52% up to 89.6%).

4.5. Porting Solvers to PERKS

Transforming the existing iterative solvers to PERKS is straightforward. This section first explains briefly how end-users can transform or port their iterative solvers to PERKS. Next, we elaborate on how we implemented memory-bound iterative methods (2D/3D stencils and a conjugate gradient solver) as PERKS.

4.5.1. Transforming Kernels to PERKS: the End-user Perspective

Identifying the minimal concurrency of the kernel

The end-user can rely on CUDA APIs [3] [85] to get the max concurrently running parameters. For even better performance, the end-user only needs to reduce the device occupancy to its minimum (while maintaining performance) via manual tuning of the kernel launch parameters or using auto-tuning tools [107–109].

[3] cudaOccupancyMaxActiveBlocksPerMultiprocessor.

Porting a Kernel to become PERKS

As Listing 4.1 shows, PERKS does not modify the computation; the manually written code to move the time loop inside the kernel and load/store to cache is straightforward. Alternatively, though outside this paper's scope, we point out the possibility of simplifying the process of converting a kernel to PERKS by using source-to-source translation, C++ templates, or domain-specific languages.

What to Cache

The end-user can use a profiler offline to decide on what data arrays to cache by identifying the arrays that generate the most traffic to/from global memory. In many iterative solvers, profiling is not even needed since the algorithm clearly implies the main data array(s) causing the highest traffic (e.g., the matrix A in conjugate gradient and the discretized domain in stencil applications).

Where to Cache

The end-user would simply use the unused shared memory for caching. For additional performance benefits, advanced users can choose to also cache in registers by manually identifying the adequate number of registers that can be used for caching, without causing register spilling (we provide a Python script to automate this process) or by following the trace of existing on-chip resources management research [110, 111]. We anticipate the possibility of automating this step by source-to-source translation or domain-specific languages so that this step of using on-chip resources could be as easy as adding a persisting range in the domain, similar, in principle, to the method of using L2 cache residency control in A100 [20], except that l2 cache residency control does not guarantee the data is definitely persistent [64].

4.5.2. Transforming Stencil Kernels to PERKS

Stencil Kernel

We use SHM [6] implementation as baseline. In SHM, 3D stencil implementation uses the standard shared memory implementation where 2D planes (1D planes in 2D stencils) are loaded one after the other in shared memory. Each thread computes the cells in a vertical direction [30,112].

Porting the Stencil Kernel

As Listing 4.1 shows, we do not interfere with compute; before the computation is started do we load the input from the registers/shared/global memory as Listing 4.2 shows; and after the computation is finished do we store the results in the registers/shared/global memory. After adjusting to handle the input and output of the computation part of the kernel, we exchange the halo region (inter thread block dependency data) between time steps. To ensure coalesced memory accesses in the halo region, we transpose the vertical edges of the halo region. Also, we can reuse the on-chip resources for caching as soon as the data is consumed. Finally, since the original kernel uses shared memory [30, 112] and registers [8] to optimize stencils, we use the version of the output

Listing 4.1: Iterative 2D 5-pt stencil implemented in PERKS.

```
__global__ void 2d5pt_PERKS(ptr_in, ptr_out){...
  for(k=0; k<timestep; k++){...
    //Use branch to control the source
    load(bid,tid,[ptr_in|sm_cache|reg_cache]->sm_in);
    2d5pt_Compute(sm_in,reg_out);
    //Use branch to control the destination
    store(bid,tid,reg_out->[ptr_out|sm_cache|reg_cache]);
    ...
    grid.sync();
    ...
  }
  ...
}

__device__ void 2d5pt_Compute(sm_in, reg_out){
    x = threadIdx.x;
    t[IPT+2]; //IPT: items per thread
    for(y=0; y< IPT+2; y++)t[y]=sm_in[x, y+ind_y-1];
    for( y=0; y< IPT; y++){
      reg_out[y]=sm_in[x+ind_x-1,y+1+ind_y] *WEST
                 +sm_in[x+ind_x+1,y+1+ind_y] *EAST
                 +t[y-1+1]                   *SOUTH
                 +t[y+1]                     *CENTER
                 +t[y+1+1]                   *NORTH;
    }
}
```

residing in shared memory or registers at the end of each time step as an already cached output. In this way, we avoid an unnecessary copy to shared memory and registers that we would use for caching.

4.5.3. Transforming the Conjugate Gradient Solver to PERKS

Conjugate Gradient Kernel

For simplicity and accessibility, we use the Conjugate Gradient (CG) solver implementation that is part of the CUDA SDK samples (*conjugateGradientMultiBlockCG* [12]). Since the implementation of SpMV in the CG sample is relatively naive, we use the highly optimized merge-based SpMV [13] that is part of the C++ CUB [91] library in the CUDA Toolkit [113], as it fits naturally with the caching scheme in PERKS. Due to the space limit, we do not discuss the details of merge-based SpMV. The reader can refer to details in [13].

Porting the Conjugate Gradient Kernel

We do not change the implementation or algorithm of the merge-based SpMV since PERKS does not necessitate changes in the underlying algorithm. For merge-based SpMV, we cache the matrix A since it is the largest data array in the solver. To improve performance further, we cache the residual vector r and the intermediate results. The merge-based SpMV [13] in CUB [91] is composed of two steps: *search* and *compute*. The search step is done twice. The search step first

Listing 4.2: Pseudocode for the load API in PERKS: The if-statement is statically determined at compile time, so its performance is akin to template specialization.

```
template<int global_index,
    bool fromReg=isInReg(global_index),
    bool fromSM=isInSM(global_index)>
void load(int bid, int tid, TILE* in_ptr,
  TILE reg_cache, TILE* sm_cache, TILE* aim_s){
  if(fromReg)
    {aim_sm[aim_index(tid)]=reg_cach;}
  else if(fromSM)
    {aim_sm[aim_index(tid)]=sm_cache[sm_cache_index(bid, tid)];}
  else
    {aim_sm[aim_index(tid)]=in_ptr[global_index];}
}
```

finds the workload for each thread block and then finds the workload for each thread inside a thread block. The search result for the thread block workloads in global memory is saved since the matrix is static throughout the entire iteration. The second search (thread-level) is conducted in shared memory. Those two steps repeatedly generate intermediate data that we cache, in addition to the matrix A. Listing 4.3 shows a code sample of PERKS-based Iterative SpMV, which can be extended to a conjugate gradient solver.

Merge-based SpMV originally uses small thread blocks, i.e., 64 threads per TB. This introduces a high volume of concurrently running thread blocks per streaming multiprocessor. To reduce the device occupancy while maintaining performance, we increased the TB size to 128 and slightly changed the memory access order to accommodate the larger TB size.

4.5.4. PERKS and CUDA Considerations

Restrictions of Synchronization APIs

PERKS relies on cooperative groups APIs [85] (supported since CUDA 9.0). Currently, the APIs do not allow over-subscription, i.e., one needs to explicitly assign workload to blocks and threads to expose enough parallelism to the device. However, it is worth mentioning that this API does not limit the flexibility, as different kernels can still run concurrently in a single GPU, as long as they as a whole don't exceed the hardware limitation.

New Features in Nvidia Ampere

The Nvidia Ampere generation of GPUs introduced two new features that could potentially improve the performance of PERKS. Namely, asynchronous copy for shared memory and L2 cache residency control [20]. We did not observe a noticeable performance difference when testing asynchronous copy to cache in PERKS. For L2 cache residency control, we experimented with setting the input and halo region to be persistent in stencils. We observed a 8% slowdown and no change in performance, respectively. Accordingly, we do not include those new features in our PERKS implementations.

Listing 4.3: Iterative Sparse matrix–vector multiplication with merge-based SpMV [13] implemented in PERKS.

```
__global__ void Iterative_SpMV_PERKS(...){
  ...
  while(...){
    //merge-based SpMV [13] contains:
    //search: determine the workload range
    //SpMV_Compute: performs SpMV within range
    ...
    //determine thread block workload by recomputing or by loading from cache
    get([cached_tb_range|search(gm_matrix)]-> tb_range);
    //load tb_range of matrix to shared memory, from global memory or from cache
    load(tb_range,[sm_matrix|gm_matrix]->tb_sm_matrix);
    ...
    //determine thread workload by recomputing or by loading from cache
    get([cached_t_range|search(tb_matrix)]->t_range);
    SpMV_Compute(tb_sm_matrix, t_range);
    ...
    grid.sync();
    ...
  }
  ...
}
```

Register pressure in PERKS

One concern with PERKS is that kernels might run into register pressure if the compiler is not optimally reusing registers for different time steps, potentially affecting concurrency and penalizing performance. To illustrate this issue, take a high register-pressure 2D 25-point double precision Jacobi stencil as an example. The shared memory optimized baseline version (SHM) uses 78 registers per thread, yet the PERKS version uses 112 registers[4]. Similar behavior is also observed in other stencil benchmarks. Reducing the occupancy while maintaining the concurrency –as mentioned in the previous section– reduces the impact of this compiler's inefficiency in register reuse in all the benchmarks we report in the results section. In the above example, at worst, 48 registers among the maximum available 178 registers per thread could not be used for caching data; it neither harms concurrency nor triggers register spilling.

4.6. Evaluation

4.6.1. Hardware and Software Setup

The experimental results presented here are evaluated on the two latest generations of Nvidia GPUs: Volta V100 and Ampere A100 with CUDA 11.5 and driver version 495.29.05.

We run each evaluation ten times for all iterative stencils and conjugate gradient experiments. All experimental results reported are done in double precision.

[4]We gathered the number of registers used by finding the maximum number of registers available as a cache before spilling with "__launch_bounds__" instruction. Register spilled can be indicated by '-Xptxas "-v -dlcm=cg"' flag.

Table 4.2: Stencil benchmarks and domain sizes we use. A detailed description of the stencil benchmarks can be found in [1, 2]. For fairness, the domain sizes we use are the minimum domain sizes that would saturate the device for different stencil benchmarks. The minimum domain sizes are identified by running each benchmark at different domain sizes and using the domain size after which the performance (in GCells/s) seize to improve

Benchmarks(Stencil Order, FLOPs/Cell)	**A100**	**V100**
2d5pt (1,10)	2304×2304	2048×1280
2ds9pt (2,18)	2304×2304	2048×1280
2d13pt (3,26)	4608×3072	2048×2048
2d17pt (4,34)	3072×2304	4096×2560
2d9pt (1,18)	2304×2304	2048×1280
2d25pt (2,50)	4608×3072	2048×1280
3d7pt (1,14)	$256 \times 288 \times 256$	$128 \times 128 \times 128$
3d13pt (2,26)	$256 \times 288 \times 256$	$256 \times 320 \times 256$
3d17pt (1,34)	$256 \times 288 \times 256$	$160 \times 160 \times 256$
3d27pt (1,54)	$256 \times 288 \times 256$	$160 \times 160 \times 256$
poisson (1,38)	$256 \times 288 \times 256$	$160 \times 160 \times 256$

4.6.2. Benchmarks and Datasets

Stencil Benchmarks

To evaluate the performance of PERKS stencils, we perform a wide set of experiments on various 2D/3D stencil benchmarks (listed in Table 4.2).

We compare PERKS (w/ SHM [6] as base) with a wide range of state-of-the-art stencil implementations/libraries: PPCG [7], Bricks [1], SSAM [8], STENCILGEN [4] and SHM [6]. The implementations/libraries represent different classes of stencil optimization approaches: code auto-generation (PPCG), vector-level data reuse (Bricks), shared memory optimization (SHM), accumulated summations optimization (SSAM), and temporal blocking (STENCILGEN). For a fair comparison, when comparing with STENCILGEN [4], SSAM [8], and Bricks [1], we use the default setting (including the default domain size) in their papers, except that we adjusted the build system to add the latest GPU (A100). We use SSAM's setting to evaluate PPCG [7].

We use the test data provided by STENCILGEN [4]. We tested three PERKS (SHM) implementations: PERKS_SM which only uses shared memory to cache data; PERKS_REG which only uses registers to cache data; and PERKS_MIX which uses both shared memory and registers to cache data. Due to space limitations, we report only the peak performance among those three PERKS variants.

Conjugate Gradient Datasets

The datasets for conjugate gradient come from the SuiteSparse Matrix Collection [3]. We selected symmetric positive definite matrices that can converge in a CG solver. The details of the

Table 4.3: Datasets for the Conjugate Gradient (CG) solver (from SuiteSparse [3])

Code	Name [3]	Rows	NNZ
D1	Trefethen_2000	2,000	41,906
D2	msc01440	1,440	46,270
D3	fv1	9,604	85,264
D4	msc04515	4,515	97,707
D5	Muu	7,102	170,134
D6	crystm02	13,965	322,905
D7	shallow_water2	81,920	327,680
D8	finan512	74,752	596,992
D9	cbuckle	13,681	676,515
D10	G2_circuit	150,102	726,674
D11	thermomech_dM	204,316	1,423,116
D12	ecology2	999,999	4,995,991
D13	tmt_sym	726,713	5,080,961
D14	consph	83,334	6,010,480
D15	crankseg_1	52,804	10,614,210
D16	bmwcra_1	148,770	10,644,002
D17	hood	220,542	10,768,436
D18	BenElechi1	245,874	13,150,496
D19	crankseg_2	63,838	14,148,858
D20	af_1_k101	503,625	17,550,675

selected datasets are listed in Table 4.3.

We compare the performance of PERKS (CG Solver) with Ginkgo library [14], a widely used library heavily optimized for GPUs (including A100). We run 10,000-time steps in our performance evaluation (similar to Ginkgo's basic setting [14]). We report the speedup per time step and the measured sustained bandwidth achieved by Ginkgo. For PERKS (CG Solver), we run different variants that implement caching the vector r or the matrix A plus the additional caching of TB-level search results and thread-level search result policies. We only report the best-performing variant for each dataset.

4.6.3. Sizes of Domains and Problems

PERKS intuitively favors smaller domain/problem sizes. However, for a fair evaluation of PERKS, we can not choose arbitrarily small domain sizes; we need domain/input sizes that fully utilize the device's computing capability. Similar to [114], we conducted an elaborate set of experiments for every stencil benchmark to identify the minimum domain size that would fully utilize the device. Note that domain/problem sizes beyond domain/problem sizes that could fully utilize the device are effectively serialized by the device once we go beyond peak concurrency sustainability by the device. Table 4.2 summarizes the domain sizes for the different stencil benchmarks that would achieve a fair performance in the SHM implementation. For conjugate gradient experiments, we

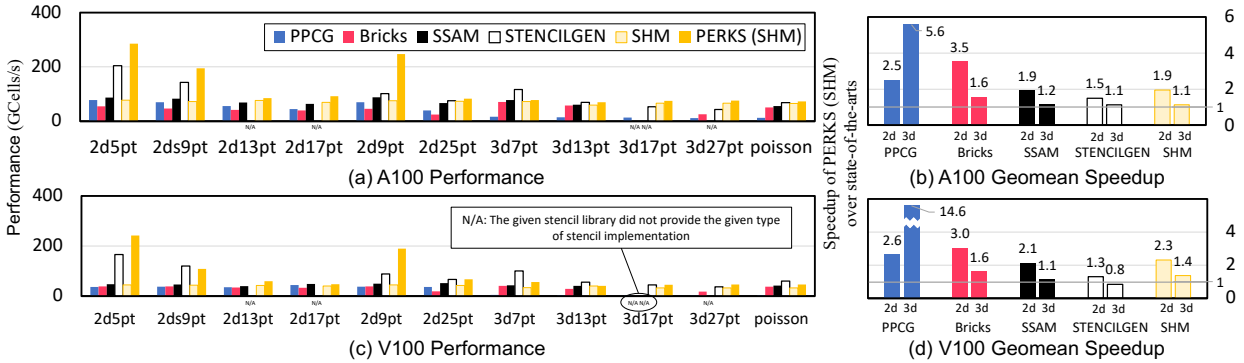


Figure 4.6: Comparison of PERKS(SHM [6]) over a wide range of stencil libraries in a wide range of 2D/3D stencil benchmarks on A100 (top) and V100 (bottom) GPUs. (a) & (c) in the left report the performance; (b) & (d) in the right report the geometric mean speedup of PERKS(SHM) over other state-of-the-art implementations.

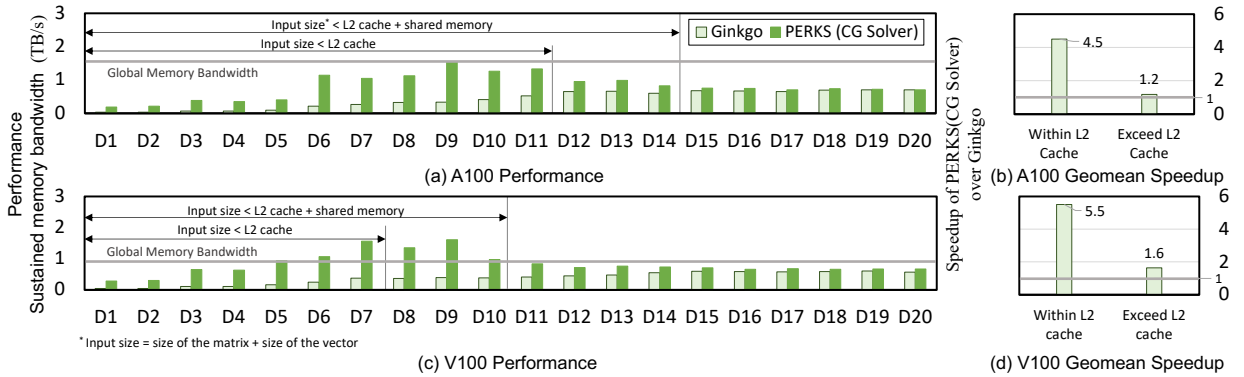


Figure 4.7: Comparison of PERKS(CG Solver, CUDA Sample [12]+CUB [13]) over Ginkgo [14] on A100 (top) and V100 (bottom) GPUs in datasets (D1 to D20) listed in Table 4.3. (a) & (c) in the left report the performance (sustained memory bandwidth); (b) & (d) in the right report the geometric mean speedup of PERKS(CG Solver) over Ginkgo.

include datasets from SuiteSpare that cover a wide range of problem sizes: from strong-scaling small dataset sizes that would fit in L2 cache and up to large dataset sizes typically reported by libraries for a single GPU of the same generations we use (Gingko [14, 115] and MAGMA [116, 117]).

4.6.4. Iterative 2D/3D Stencils

Figure 4.6 compares the performance of SHM [6] and PERKS-based SHM with a wide range of state-of-the-art stencil implementations/libraries. The performance of SHM is comparable to state-of-the-art implementations, i.e., Bricks [1] and SSAM [8], across all stencil benchmarks. Applying PERKS consistently speedup SHMs: compared to SHM, PERKS (SHM) achieves a geometric mean speedup of 1.95x in A100 and 2.31x in V100 for 2D stencils. The geometric mean speedup for 3D stencils is 1.13x for A100 and 1.36x for V100.

Compared to the best state-of-the-art spatial blocking implementations/libraries, SSAM, and

PERKS (SHM) achieve a geometric mean speedup of 1.91x and 2.11x for 2D stencils in A100 and V100, respectively. The geometric mean speedup for 3D stencils is 1.15x for A100 and 1.14x for V100. In comparison to the state-of-the-art temporal blocking implementation of STENCILGEN, PERKS (SHM) achieves a geometric mean speedup of 1.28x (A100) and 1.02x (V100).

4.6.5. Conjugate Gradient

Figure 4.7 compares PERKS (CG Solver) to Ginkgo. We observe a significantly higher performance advantage when the input size is within the L2 cache capacity. This phenomenon implies that PERKS (CG Solver) automatically benefits from the large L2 cache, possibly because the constant matrix can reside in the L2 cache, saving memory traffic to the global memory. When the input is less than the L2 capacity, PERKS running on A100 achieves a geometric mean of 4.49x speedups; in V100, PERKS achieves 5.50x speedups. When the input matrix exceeds the L2 cache capacity, PERKS running on A100 achieves a geometric mean of 1.18x speedup. On V100, PERKS achieves a geometric mean of 1.64x (double) speedups. It is important to remember that the Ginkgo library that we use as a baseline is among the top-performing libraries in CG solvers, emphasizing GPU optimizations [14].

Note that regardless of whether we stay within the L2 cache capacity or exceed it, we are still caching the domain using one of the caching policies we described earlier.

4.7. Discussion

4.7.1. Discussion of the Results

We want to emphasize that for large problem sizes, PERKS achieves very high performance. To illustrate it, as can be deduced from Figure 4.6, By applying PERKS in V100, the geometric mean speedup is 1.81x, which is $1.20x$ of what one generation of hardware improvements in A100 provide (1.51x). Similarly, in the conjugate gradient solver with the large datasets D15-D20 (Figure 4.7), by applying PERKS in V100, the geometric mean speedup is 1.15x, which is 98.29% of the improvements A100 can provide (1.17x). Finally, since the results for small problems are outstanding, we report the results for smaller problem sizes separately to emphasize the large extent to which PERKS would be beneficial for strong scaling.

4.7.2. Hardware difference

PERKS performs better in V100 than in A100. This is due to that 1). Global memory bandwidth in A100 is 1.73x faster than V100, which results in a faster baseline in A100 compared to V100; 2). The on-chip resources in A100 are only 1.61x larger than V100, limiting the relative speedup of PERKS in A100.

4.7.3. Small Domain

PERKS favors small domains. We also conduct a test for stencils with domain sizes that perfectly fit into on-chip resources. The results are presented in Figure 4.8. Applying PERKS

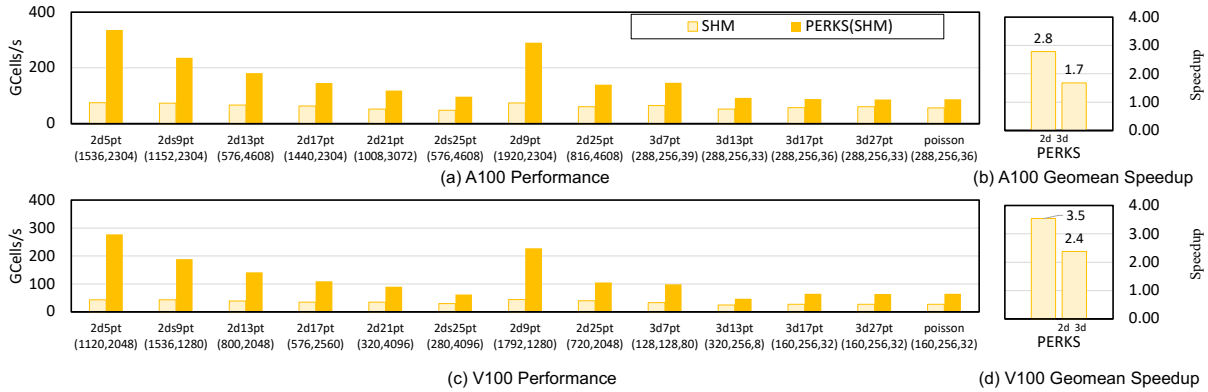


Figure 4.8: PERKS in small domain size

further speedup SHMs: in comparison to SHM, PERKS (SHM) achieves a geometric mean speedup of 2.78x (from 1.95x) in A100 and 3.54x (from 2.31x) in V100, for 2D stencils. The geometric mean speedup for 3D stencils is 1.68x (from 1.13x) for A100 and 2.38x (from 1.36x) for V100. Extending the performance of a small domain to any general cases belongs to the topic of temporal blocking. We will cover this topic in Chapter 5.

4.7.4. Caching

Caching to Shared Memory, Registers, or Both?

Figure 4.9 shows the performance of stencil benchmarks when using different resources for caching. The intuition is that using both shared memory and registers would always be better (more cache-able space). The results show that this is usually the case. There can however be exceptions. For instance, in our observations (that we don't include due to space limits) we see that for higher-order stencils, using shared memory and registers is often not the ideal choice (presumably due to arising register pressure).

What to Cache?

We highlight important observations when varying the data to cache in the conjugate gradient solver (see Figure 4.10). First, the implicit cache policy (IMP) achieves a geometric mean of 3.61x and 1.19x over Ginkgo for data sets size both within and (surprisingly) when exceeding L2 cache, respectively. This means **PERKS can gain speedup before applying any explicit caching policy** by getting hits in the L2 cache. Second, the speedup difference between caching the vector (VEC) or not is usually insignificant for most situations. This might be because vectors are generally not large enough to consume all available cache resources. Third, as expected, the general tendency is that the more PERKS caches, the more speedup there is. So we generally get the highest speedup by caching the matrix (MAT) or caching both the vector and matrix (MIX). The exception is in the case of single precision when the dataset sizes exceed L2 cache: we still get speedup from MAT and MIX, albeit at lower rates than when the data is within the L2 cache.

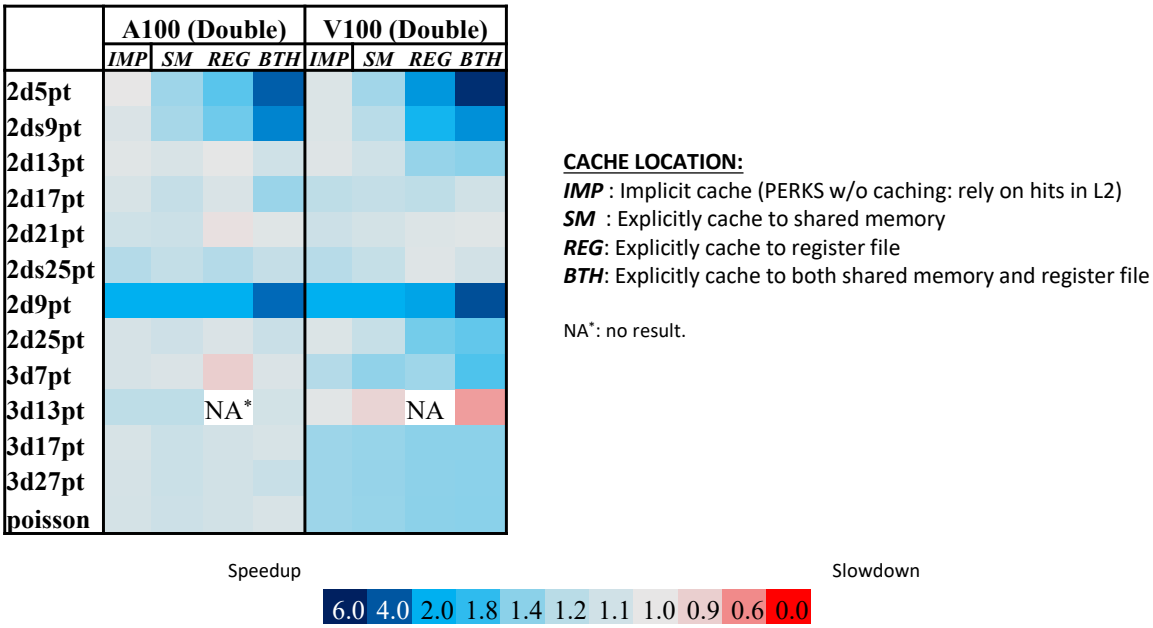


Figure 4.9: Using different resources for caching stencils.

What Should the End-user Do?

In summary, the cache policy analysis for conjugate gradient (which is a fairly complex solver) shows that a simple greedy approach of targeting the largest data arrays in as many caching resources as possible gives mostly the best performance. While there can be outliers, the simple greedy policy is in most cases effective enough and also simple for end-users (since they only need to identify the arrays generating the most traffic).

4.7.5. Hit Rate

To understand how PERKS influence hit rate, we present case studies involving 2D 5-Point Jacobian stencils (representing 2D stencils) and 3D 7-Point stencils (representing 3D stencils). The results are presented in Table 4.4, leading to the following observations:

Due to PERKS's leverage of larger scratchpad memory, which leads to a smaller L1 cache capacity in GPUs, applying PERKS might reduce the hit rate of the L1 cache. This could be an issue in 3D stencils, as the hit rate is reduced by 27% in A100 and 35% in V100. This partially explains why PERKS does not perform equally well in 3d stencils compared to 2d stencils.

On the other hand, PERKS increases the cache hit rate of the L2 cache. Part of the hit rate improvement may come from the better memory access pattern of the halo region. The remaining increase in hit rate may result from reduced necessary domain access (reducing the denominator).

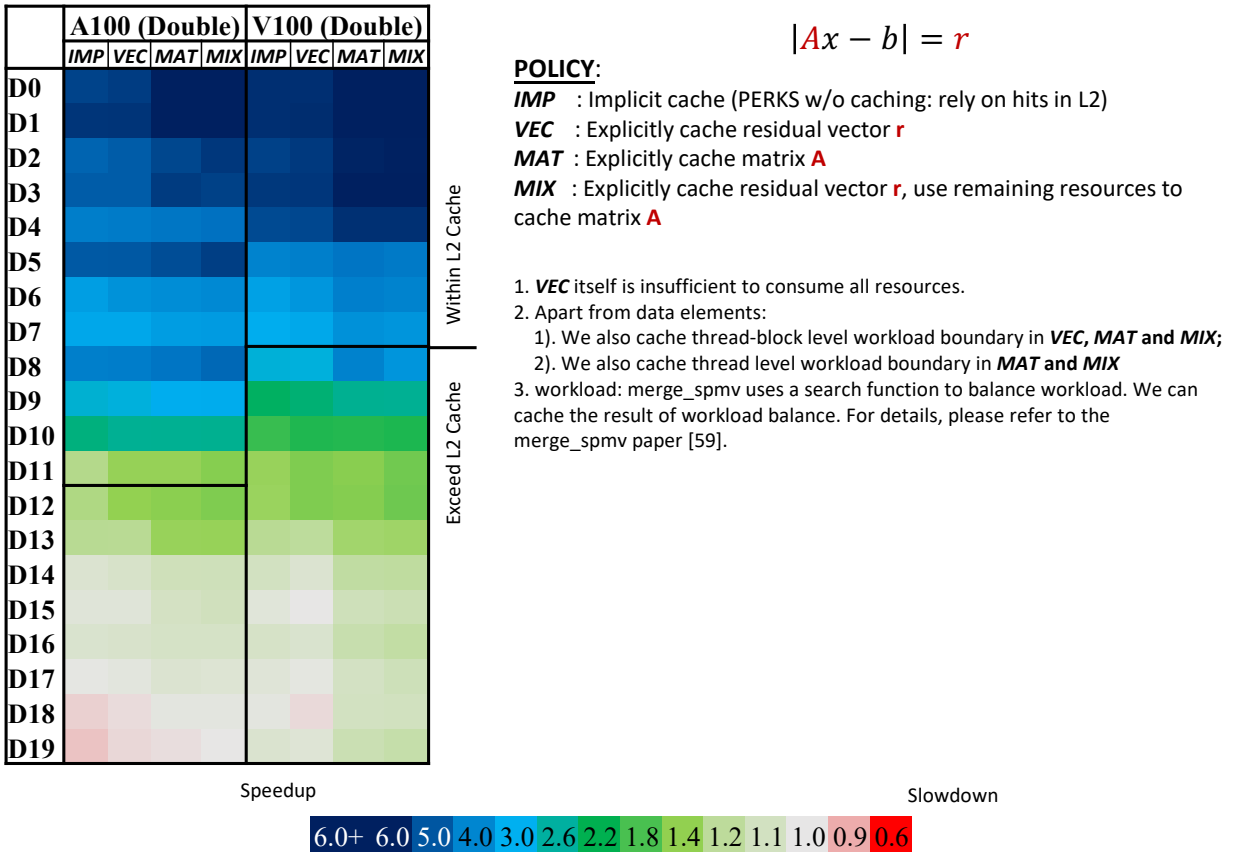


Figure 4.10: Caching different data for the conjugate gradient.

4.8. Related Work

The concept of persistent threads and persistent kernels dates back to the introduction of CUDA [118, 119]. The main motivation for persistence at the time was load imbalance issues with the runtime warp scheduler [118,120]. Later research focused on using persistent kernels to overcome the kernel invocation overhead (which was high at the time). GPUrdma [121] and GPU-Ether [122] expanded on the concept of persistent kernels to reduce the latency of network communication.

As on-chip resources increased, researchers began to capitalize on data reuse in the persistent kernel. Most of them focused on specific applications, GPUrdma [121] proposed to keep the constant matrix in shared memory. Khorasani et al. [123] proposed to keep parameters in registers. Zhu et al. [124] proposed a sparse persistent implementation of recurrent neural networks. To our knowledge, this work is the first to propose a generic and methodological blueprint for accelerating memory-bound iterative applications using persistent kernels.

Table 4.4: Hit rate of 2D 5-Point Jacobian stencils and 3D 7-Point stencils in V100 and A100 before and after applying PERKS

L1 Cache Hit Rate	**Implementation**	**2d5pt**	**3d7pt**
V100	**SHM**	6.48%	15.90%
	PERKS(SHM)	6.63% (+2.3%)	10.39% (-34.7%)
A100	**SHM**	6.61%	9.51%
	PERKS(SHM)	5.88% (-11.0%)	6.94% (-27.0%)
L2 Cache Hit Rate	**Implementation**	**2d5pt**	**3d7pt**
V100	**SHM**	52.30%	58.60%
	PERKS(SHM)	87.32% (+67.0%)	61.73% (+5.3%)
A100	**SHM**	65.81%	66.06%
	PERKS(SHM)	85.89% (+30.5%)	66.55% (+0.7%)

4.9. Conclusion

We propose a persistent kernel execution model for iterative applications. We enhance performance by moving the time loop to the kernel and cache the intermediate output of each time step with unused on-chip resources. We show a notable performance improvement for iterative 2D/3D stencils and a conjugate gradient solver for both V100 and A100 over highly optimized baselines.

CHAPTER 5

EBISU: Epoch Blocking for Iterative Stencils, with Ultracompact parallelism

Iterative stencils are used widely across the spectrum of High Performance Computing (HPC) applications. Many efforts have been put into optimizing stencil GPU kernels, given the prevalence of GPU-accelerated supercomputers. To improve the data locality, temporal blocking is an optimization that combines a batch of time steps to process them together. Under the observation that GPUs are evolving to resemble CPUs in some aspects, we revisit temporal blocking optimizations for GPUs. We explore how temporal blocking schemes can be adapted to the new features in the recent Nvidia GPUs, including large scratchpad memory, hardware prefetching, and device-wide synchronization. We propose a novel temporal blocking method, EBISU, which champions low device occupancy to drive aggressive deep temporal blocking on large tiles that are executed tile-by-tile. We compare EBISU with state-of-the-art temporal blocking libraries: STENCILGEN and AN5D. We also compare EBISU with state-of-the-art stencil auto-tuning tools equipped with temporal blocking optimizations: ARTEMIS and DRSTENCIL. Over a wide range of stencil benchmarks, EBISU achieves speedups up to 2.53x and a geometric mean speedup of over 2.0x speedup over any state-of-the-art libraries and implementation. EBISU are available at https://github.com/neozhang307/EBISU-ICS23.

The contributions of this chapter are as follows:

- We propose the design principle of EBISU: low-occupancy execution of a single tile at a time while scaling the use of resources to improve data locality (Section 5.3).
- We introduce practical resource-scaling schemes that efficiently improve performance (Section 5.4 and Section 5.5).
- We include an analysis of the practical attainable performance (Section 5.6) to support the design decisions (Section 5.7) for EBISU. We build on our analysis to identify how various factors contribute to the performance of EBISU.
- We evaluate EBISU across a wide range of stencil benchmarks (Section 5.8). Our implementation achieves over 2.0x speedup over any state-of-the-art libraries and implementation. And we achieve a geomean speedup of 1.53x over the top performing state-of-the-art implementations for each stencil benchmark.

5.1. Introduction

Stencils are patterns in which a mesh of cells is updated based on the values of the neighboring cells. They are common computational patterns that exist widely in many scientific applications. And they account for up to 49% of workloads in many HPC centers [34]. Applications of stencils include mainly finite difference solvers of Partial Differential Equations. (PDEs) [125, 126]. PDEs further support a wide spectrum of applications, spanning from weather modeling and seismic simulations to fluid dynamics simulations [127].

Many efforts have gone into optimizing stencils [46, 49, 72]. Due to the low computational intensity of stencils [128], combining steps and processing them together, i.e., temporal blocking, is an optimization widely used in iterative stencils [46, 48, 129–131]. This optimization increases the computational intensity, which comes at the price of adopting complex schemes to handle the constraints of temporal dependencies. Traditionally, temporal blocking resolves the dependency between time steps either by redundant overlapping of tiles [4, 5, 72, 74] or by complicated titling geometry (e.g., diamond [54] and hexagonal [132, 133]). Either way, the overhead of resources for resolving the temporal blocking dependencies increases the data with the depth of temporal blocking [5]. An increasing number of time steps to block gradually moves the kernel's bottleneck from the memory throughput to be bound by either the memory latency or register pressure [5]. Among temporal blocking optimization efforts, many of them are related to specific hardware, e.g., FPGA [131,134], CGRA [135], multi-core [136], and many-core [4] architectures. This paper focuses on GPUs' prevalence in HPC systems [50].

There are notable changes in key features when closely observing the latest GPUs. We observe a significant increase in cache capacity. Specifically, the total capacity of the user-managed cache (shared memory) increased from 720 KB in K20 [80] to 17, 712 KB in A100 [20]. The shared memory capacity has increased by 24.6x in recent decades. In addition, GPUs provide features that CPUs have supported for years. Examples include cooperative groups (i.e., device-wide barriers), low(er) latency of operations, and asynchronous copy of shared memory (i.e., prefetching) [64].

These recent advancements present new opportunities for implementing aggressive optimizations specifically designed for stencil kernels. However, existing state-of-the-art temporal blocking implementations, e.g., AN5D [5] and STENCILGEN [4], are designed to run at high occupancy and are hence relatively conservative in the use of resources to avoid adverse pressure on resources (ex: register spilling). For example, AN5D [5] uses a maximum of 96 registers per thread and STENCILGEN [4] uses a maximum of 64 registers per thread for all the benchmarks reported. Yet the limit for registers is 255 in both V100 and A100 [64] GPUs. For shared memory usage, AN5D [5] consumes at most 34.8 MB per thread block, and STENCILGEN [4] uses at most 33.8 MB per thread blocks. Yet the limit for shared memory is 164 MB in A100 [64] GPUs. This conservative manner is partly due to the intention to ensure a higher occupancy.

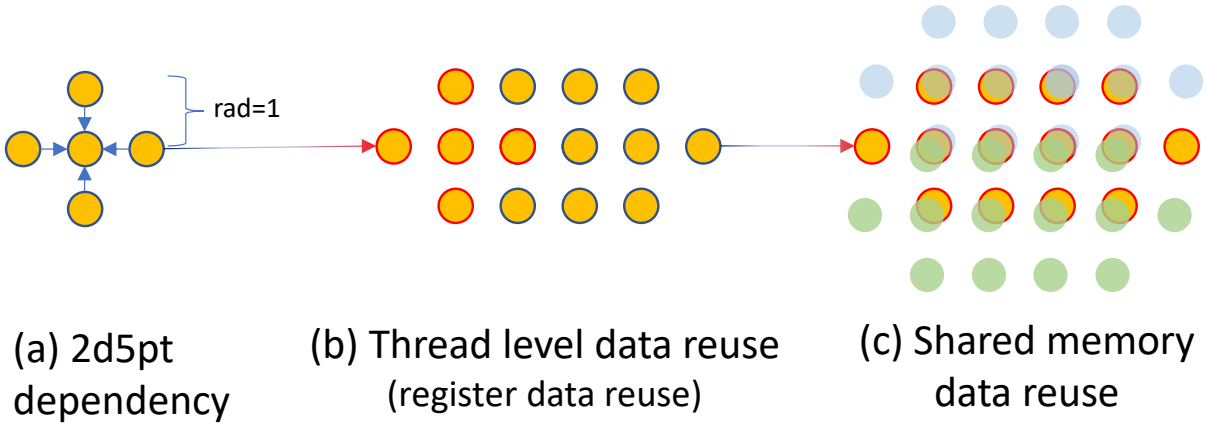


Figure 5.1: Spacial Blocking, using 2D 5-point Jacobian (2d5pt) stencil as an example

In this chapter, we take inspiration from the work of Volkov et al. [65]; we propose a different approach to occupancy and performance in temporal blocking. We first determine a parallelism setting that is minimal in occupancy while sufficient in instruction-level parallelism. We base our approach for temporal blocking on lower occupancy, i.e., we build large tiles running at the minimum possible concurrency to be executed tile-by-tile and accordingly scale up the use of on-chip resources to run the tile at the maximum possible performance.

We propose *EBISU*: Epoch (temporal) Blocking for Iterative Stencils, with Ultracompact parallelism. EBISU's design principle is to run the code at the minimum possible parallelism to saturate the device and then use the freed resources to scale up the data reuse and reduce the dependencies between tiles. Though the idea is seemingly simple, the challenge is the lack of design principles to achieve scalable optimizations for temporal blocking. In other words, temporal blocking schemes in literature are designed to avoid pressure on resources since resources are scarce in over-subscribed execution; EBISU, on the other hand, assumes ample resources that are freed due to running in low occupancy and the goal is to scale the data reuse to all the available resources for a single tile at a time that spans the entire device. We drive EBISU through a cost model that decides how to scale resource use at low occupancy effectively.

5.2. Background

5.2.1. Stencils

Stencils are characterized by their memory access patterns. We present the pseudo codes for the 1D 3-Point, 2D 5-Point and 3D-7-Point Jacobian Stencil in Listing 5.1, Listing 5.2, and Listing 5.3 respectively. We use a 2D Jacobian 5-point (2d5pt) stencil as an example. Figure 5.1.a illustrates the neighborhood dependencies of the 2d5pt stencil. To compute one point, the four adjacent points are necessary. Two blocking methods are widely used to optimize iterative stencils for data locality:

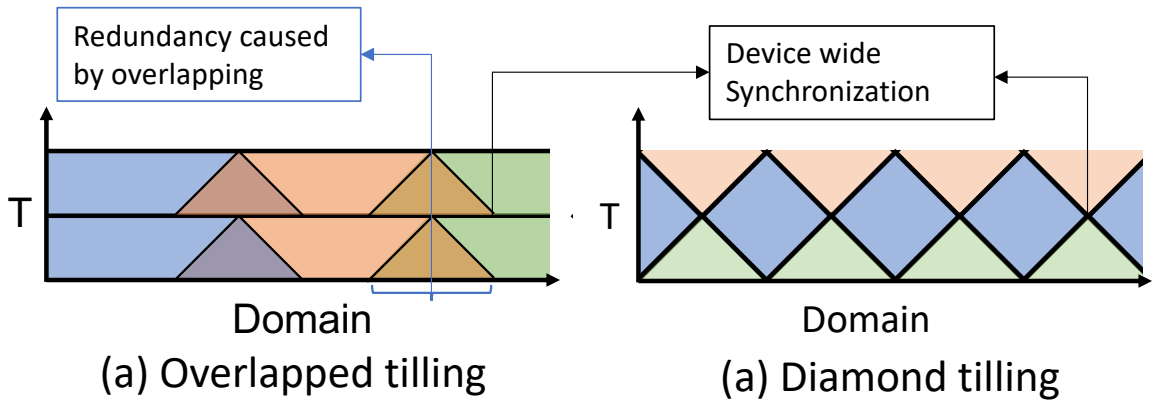


Figure 5.2: Temporal Blocking

Spatial Blocking

In spatial blocking on GPUs, thread (blocks) load a single tile of the domain to its local memory to improve the data locality among adjacent locations [70, 71]. The local memory can be registers [1, 8](Figure 5.1.b) and scratchpad memory [6, 137](Figure 5.1.c). However, halo layer(s) are still unavoidable.

Temporal Blocking

In iterative stencils, each time step depends on the result of the previous time step. An alternative optimization is to combine several time steps to expose temporal locality [2, 5]. In this case, the temporal dependency is resolved by overlapped tilling [53, 72, 74] (Figure 5.2.a) or by applying complex geometry [55, 56] (Figure 5.2.b, diamond tiling [54, 133] as an example). The main shortcoming of the overlapped tiling is redundant computation, while the main disadvantage of complex geometry is adverse effects on cache hits [48]. Additionally, complex geometry is penalized by the device-wide synchronization necessary to ensure the result is updated in the global memory.

N.5-D Temporal Blocking

N.5-D blocking [4, 5, 138] is a combination of spatial blocking and overlapped temporal blocking [138]. Take 3.5-D temporal blocking as an example. We do spatial tiling in the X and Y dimensions and then stream in the Z dimension (2.5-D spatial blocking). As we stream over the Z dimension, each XY plane would conduct a series of temporal steps (1-D temporal blocking). This method reduces the overhead of redundant computations in an overlapped temporal blocking schema in the stream direction.

5.3. EBISU: High Performance Temporal Blocking at Low Occupancy

In this section, we give an overview of our temporal blocking method: EBISU (Figure 5.3 gives an overview) The design of EBISU follows two main principles: minimal parallelism that would saturate the device (the Minimal Parallelism step in Figure 5.3), and scalability in using resources

Listing 5.1: Pseudocode for 1D 3-Point Jacobian Stencil

```
for(int i=0; i<N; i++)
    out[i]=a*in[i-1]+b*in[i]+c*in[i+1];
```

Listing 5.2: Pseudocode for 2D 5-Point Jacobian Stencil

```
for(int i=0; i<N; i++)
    for(int j=0; j<M; j++)
        out[i][j]=a*in[i-1][j]+b*in[i][j]+c*in[i+1][j]
                 +d*in[i][j-1]]+e*in[i][j+1];
```

Listing 5.3: Pseudocode for 3D 7-Point Jacobian Stencil

```
for(int i=0; i<N; i++)
    for(int j=0; j<M; j++)
        for(int k=0; k<L; k++){
            out[i][j][k]= a*in[i-1][j]   [k];
            out[i][j][k]+=b*in[i]   [j]   [k];
            out[i][j][k]+=c*in[i+1][j]   [k];
            out[i][j][k]+=d*in[i]   [j-1][k];
            out[i][j][k]+=e*in[i]   [j+1][k];
            out[i][j][k]+=f*in[i]   [j]   [k-1];
            out[i][j][k]+=g*in[i]   [j]   [k+1];}
```

(the Implementation step in Figure 5.3). Additionally, EBISU relies on a comprehensive analysis for implementation decisions (the pink steps in Figure 5.3).

5.3.1. Saturating the Device at Minimal Parallelism

In EBISU we first tune the parallelism exposed in the kernel to find the minimal combination of occupancy and instruction level parallelism that would saturate the device. The minimal occupancy that we aim for in this paper is 12.5% since further reducing the occupancy for memory-bound codes can start to regress the performance [20]. We aim to maximize resources for increasing locality. We use Little's Law to identify the minimum parallelism (occupancy) in the code (discussed in Section 5.7.1). We point out that readers can also rely on auto-tuning tools to empirically figure out the minimal parallelism [107, 108, 137].

5.3.2. Scaling the Use of Resources

Despite the relatively large amount of on-chip resources, there is a lack of design principles that are able to scale up to take advantage of the large on-chip resources in temporal blocking. We thereby build on a set of existing optimizations to drive a resource-scalable scheme for increasing locality (Section 5.4).

5.3.3. Implementation Decisions

We base the decision to implement EBISU on our analysis of the practical attainable performance (Section 5.6). We utilize this analysis is to decide whether to implement a device tile (Section 5.7.3) and the parameterization of spatial and temporal blocking (Section 5.7.4)).

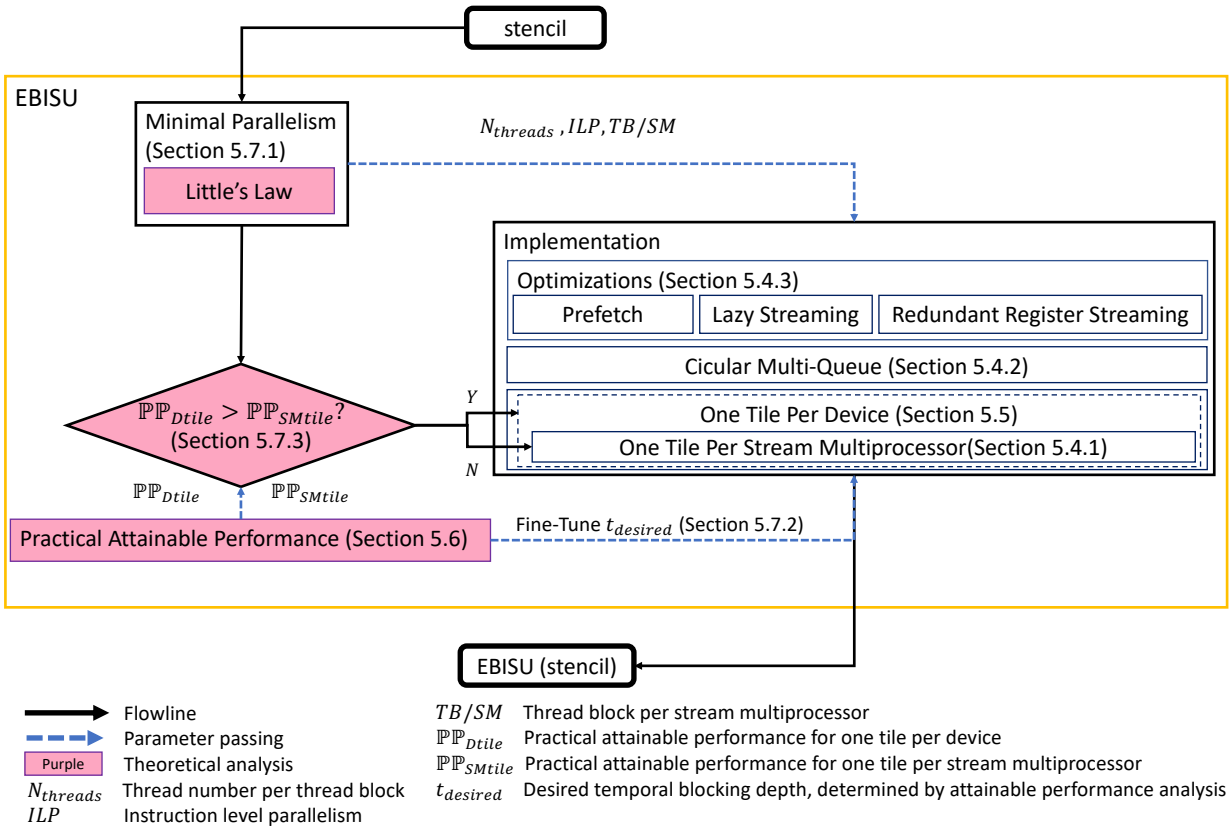


Figure 5.3: Overview of EBISU.

5.3.4. Fine-Tuning

After identifying the ideal tiling scheme and parameterization, implementation, we fine-tune the kernel to extract additional performance. For instance, we tune the temporal blocking depth (Section 5.7.2).

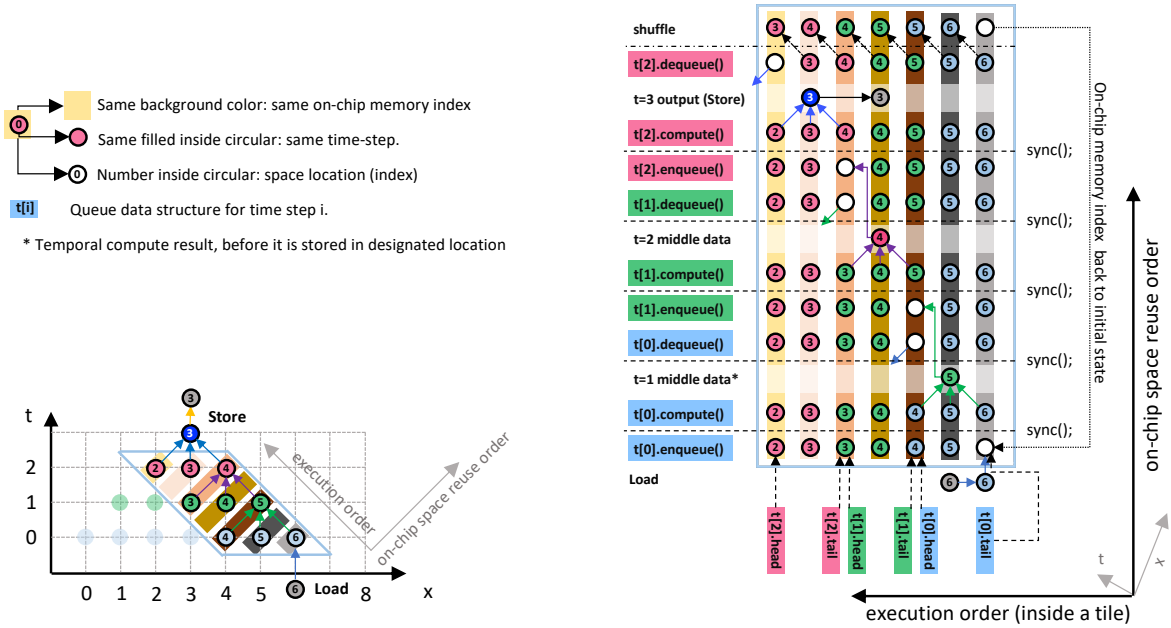
5.4. Efficiently Scaling the Use of Resources

5.4.1. One Tile At A Time

Beyond the point where the GPU becomes saturated, the workload will inevitably be serialized. We intentionally introduce a method to serialize the execution of tiles, where each tile becomes large enough to saturate the Stream Multiprocessor. We call this *SM tiling*.

5.4.2. Circular Multi-Queue

EBISU aims to scale up resource usage. One way to achieve this goal is to scale up to deep temporal blocking. This section introduces a simple data structure that efficiently manages very deep temporal blocking: namely, *circular multi-queue*. We elaborate on our design by first introducing *multi-queue* for streaming, and then we describe the implementation of the *circular multi-queue*.



(a) Temporal blocking, parallelogram denotes tiling in time and space dimensions.

We mark the execution order inside a tile, that allows space with data not used in the next tile (e.g., ⑤) to be repurposed for storing result (e.g., ④) to be used in the subsequent tiles.

(b) Using multiple queues (multi-queue) to manage temporal blocking tiling (parallelogram in (a))

Data expiration and storage in each time step can be abstracted as dequeue and enqueue operations, allowing being managed by a queue data structure. The multi-queue connects multiple queues, facilitating deep temporal blocking, seamless time step transitions, and efficient on-chip memory use.

Figure 5.4: The multi-queue data structure enables efficient temporal blocking tiling: a 1D 3-Point Jacobian stencil with a depth of 3 as an example. Figure (a) illustrates streaming with a parallelogram that we process in Figure (b). Figure (b) illustrates how queue data structure can enhance the tiling processing depicted in Figure (a). The execution order and data reuse are marked in both figures.

Multi-Queue

We use the 1D 3-Point Jacobian stencil (Listing 5.1) to illustrate our implementation. Streaming is a typical method to implement temporal blocking [4]. Figure 5.4.a demonstrates an example of streaming. The parallelogram in the figure represents the tiling in time and spatial dimensions that we process in Figure 5.4.b. The process of each time step can be abstracted as two functions: *enqueue* and *dequeue*, which are standard methods in a queue data structure. We additionally add *compute* for stencil computation. As such, we manage each time step with a queue data structure. Next, we link queues in different time steps together to become a multi-queue data structure. The data structure description and the pseudocode for multi-queue are in Listing 5.4 and Listing 5.5.

Multi-queue facilitates seamless transitions between time steps. The dequeue operation (data expiration) for the current time step runs concurrently with the enqueue operation for the next time step. After the execution of a single tile, we reset the multi-queue to its initial state - a process

Listing 5.4: Pseudocode for naive multi-queue data structure with 1D 3-Point Jacobian stencil

```
1  struct Queue {
2      REAL* d; //data array
3      index tl; //tail
4      index hd; //head
5      Queue(REAL*data,index head,index tail):
6          d(data),hd(head),tl(tail){}
7      REAL dequeue(){}//Automatically accomplished by shuffle
8      void enqueue(REAL input){d[tl]=input;}
9      REAL compute(){//1d3pt stencil
10         return a*d[hd]+b*d[hd+1]+c*d[hd+2];
11     }
12 };
13 template<int depth>
14 struct MultiQueue{
15     //Multi-queue data structure
16     REAL* d; //data array
17     index hds[depth]; //head of queues
18     index r;//range of multi-queue
19     index q_r;//range of single queue, reserved for lazy streaming
20     MultiQueue(REAL*data, index range, index queue_range):
21         d(data),r(range),q_r(queue_range)
22         for(t=0; t<depth; t++)hds[t]=queue_range-q_r*s;
23     }
24     MultiQueue(REAL*data, index range): MultiQueue(data,range,2){}
25     Queue operator[](int t) {
26         return Queue(d,hds[t],hds[t]+q_r);
27     }
28     void shuffle(){
29         //default, move data
30         sync();
31         for(int i=0; i<r-1; i++)
32             d[i]=d[i+1];
33         sync();
34     }
35 }
```

we refer to as 'shuffle.' A standard method of conducting a shuffle involves shifting values to their designated locations, as demonstrated in lines 31-38 of Listing 5.4.

It is important to note that although we base our analysis on a 1D stencil example in this section, it can be extended to 2D or 3D stencils by replacing the 1D circular points (domain cells) in Figure 5.4 to 1D lines (corresponding to 2D stencils) or 2D planes (corresponding to 3D stencils). Additionally, we can trade the concurrently processed domain cells for additional instruction level parallelism (ILP), which might be required by the parallelism setting (discussed in Section 5.7.1).

Circular Multi-Queue

We further adapt the multi-queue to be circular. We wrap the tail of time step 0 and the head of the deepest time step together. We detail the implementation of different circular multi-queue we use as follows:

Shifting Addresses: In this scheme, we only copy the index to the same place after processing a

Listing 5.5: Pseudocode for applying naive multi-queue data structure to a 1D 3-Point Jacobian stencil with temporal blocking depth of 3.

```
1  #define RANGE (7)
2  __global__ void 1d3ptstencil(REAL* input, REAL* output,...) {...
3      REAL buffer[RANGE];
4      MultiQueue t<3>(buffer, RANGE, 2);
5      for (...) {...
6          t[0].enqueue(Load(input));
7          sync();
8          for(s=0; s<3-1; s++){
9              tmp=t[s].compute();
10             sync();
11             t[s+1].enqueue(tmp);
12             //Do t[s].dequeue() t[s+1].enqueue()
13             sync();
14         }
15         Store(ouput[], t[3-1].compute()...);
16         ...
17         t.shuffle();//shuffle head and tail index for next tiling
18         ...
19     }
20 }
```

tile (at the 'shuffle step').

Computing Address: Shifting addresses is the simplest way to manage the circular data structure. However, shifting can create a long chain of dependencies as the address range increases. An alternative solution is to compute the target address (Listing 5.6 line 13.). The modulo operator (

$$\%$$

) is one of the solutions; however, this operator is time-consuming. Instead, we extend the ring index to be $range = 2^n, n \in \mathbb{Z}^+$. In this case, we have $index\%range = index\&(range - 1)$. This might consume additional space (Listing 5.6 line 33).

5.4.3. Optimizations

Prefetching

Prefetching is a well-documented optimization. Readers can refer to [137] for hints. The new asynchronous shared memory copy API offers another approach for prefetching, with a trade-off of requiring additional shared memory space for buffering.

Lazy Streaming

The naive implementation shown in Figure 5.4 and Listing 5.4 clearly suffer from the overhead of frequent synchronization. We propose *lazy streaming* to alleviate this type of overhead. The basic idea is that we delay the processing of a domain cell until all domain cells required to update the current cell are already updated. Otherwise, we would pack the planes that include the current

Listing 5.6: Pseudocode for applying circular multi-queue data structure, which inhirits the structure described in Listing 5.4.

```
index mod(index a,index r)
{
    return (r&(r-1))==0?a&(r-1):a%r;
}
struct Circular_Queue: public Queue
{
    //Circular queue inhirit from queue
    index r;
    Circular_Queue(REAL*data,index head,index tail,index range,):
        d(data),hd(head),r(range),tl(tail){}
    REAL compute()
    {
        //override 1d3pt stencil
        return a*d[hd]+b*d[mod((hd+1),r)]+c*d[mod((hd+2),r)];
    }
};
template<int depth>//Circular multi-queue inhirit from multi-queue
struct Circular_MultiQueue: public MultiQueue<depth>
{
    Circular_MultiQueue(REAL*data, index range)
        :Multi-queue<depth>(data,range){}
    Circular_MultiQueue(REAL*dat, index ran, index q_ran)
        :MultiQueue<depth>(dat,ran,q_ran){}
    Circular_Queue operator[](int t)
    {
        return Circular_Queue(d,hds[t],mod(hds[t]+2,r),r);
    }
    void shuffle()//Override shuffle for computing address schema
    {
        for(int i=0; i<r; i++)
            hds[i]=mod((hds[i]+1),r);
    }
}
#define RANGE (8)
...//kernel code unchange
```

domain cell and cache it in on-chip memory. As Figure 5.5 shows, the computation of *location 3* is postponed until the three points of the previous time steps have been updated.

The benefit of using lazy streaming is not significant in 1D stencils. In 2D or 3D stencils, we replace the points in Figure 5.5 with 1D or 2D planes for 2D or 3D stencils. The planes usually involve inter-thread dependency, which makes synchronizations unavoidable (warp shuffle when using registers for locality [8,9] or thread block synchronization when using shared memory for locality [6]). when applying *device tiling* (Section 5.5), device (grid) level synchronization becomes unavoidable, and it has a higher overhead in comparison to thread block synchronizations. As illustrated in Listing 5.7, lazy streaming can ideally reduce the synchronization to one synchronization per tile. The benefit of lazy streaming comes from the number of synchronization it reduced.

It's worth noting that double-buffering [4,5] can be viewed as a special case of lazy streaming when only a single queue evolved.

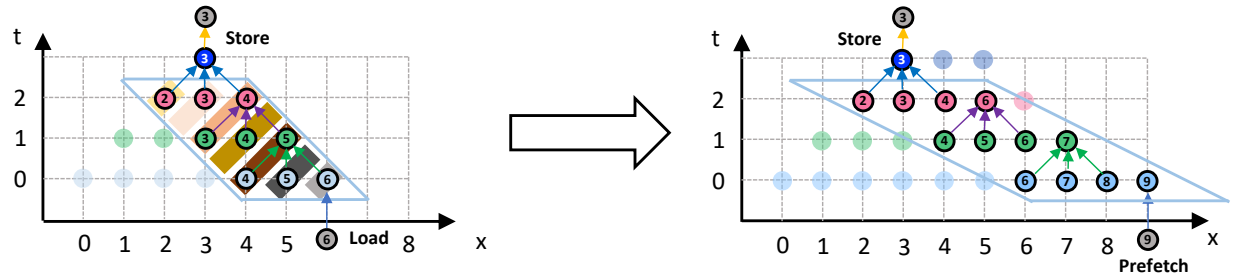


Figure 5.5: Applying lazy streaming for temporal blocking. 1D 3-Point Jacobian stencil with depth=3 as an example. Notations are the same as Figure 5.4. Additional buffer space is required to store intermediate data.

Listing 5.7: Pseudocode for applying lazy streaming to a 1D 3-Point Jacobian stencil with temporal blocking depth of 3.

```
//Lazy streaming kernel code
__global__ void 1d3ptstencil_lz(REAL* input, REAL* output,...) {...
    REAL buffer[16];//more space for buffering
    Circular_MultiQueue t<3>(buffer, 16, 3);
    for(...) {...
        t[0].enqueue(Load(input));//prefetch
        for(s=0; s<3-1; s++){
            tmp=t[s].compute();
            t[s+1].enqueue(tmp);
        }
        Store(ouput[],t[3-1].compute()...);
        ...
        t.shuffle();//shuffle head and tail index for next tiling
        sync();//One sync per tile
        ...
    }
}
```

Redundant Register Streaming

The above discussions, which do not specify the on-chip memory type, can apply to both shared memory-based and register-based implementations. However, there is one exception: the circular multi-queue cannot be implemented with register arrays since register addresses cannot be determined at compile time.

At low occupancy, we obtain a large number of registers and shared memory per thread. Therefore, by reducing the occupancy, we can afford to redundantly store intermediate data in both the registers and the shared memory. Streaming w/ caching in shared memory is discussed in STENCILGEN [4]. Streaming w/ caching in the registers is discussed in AN5D [5]. We benefit from both components by caching in both shared memory and registers. We can reduce shared memory access times to their minimum by using registers first (in comparison to AN5D) and reducing the necessary synchronizations when using only shared memory (in comparison to STENCILGEN). Additionally, due to data being mostly redundant, we can tune to reduce resource usage in either part of registers

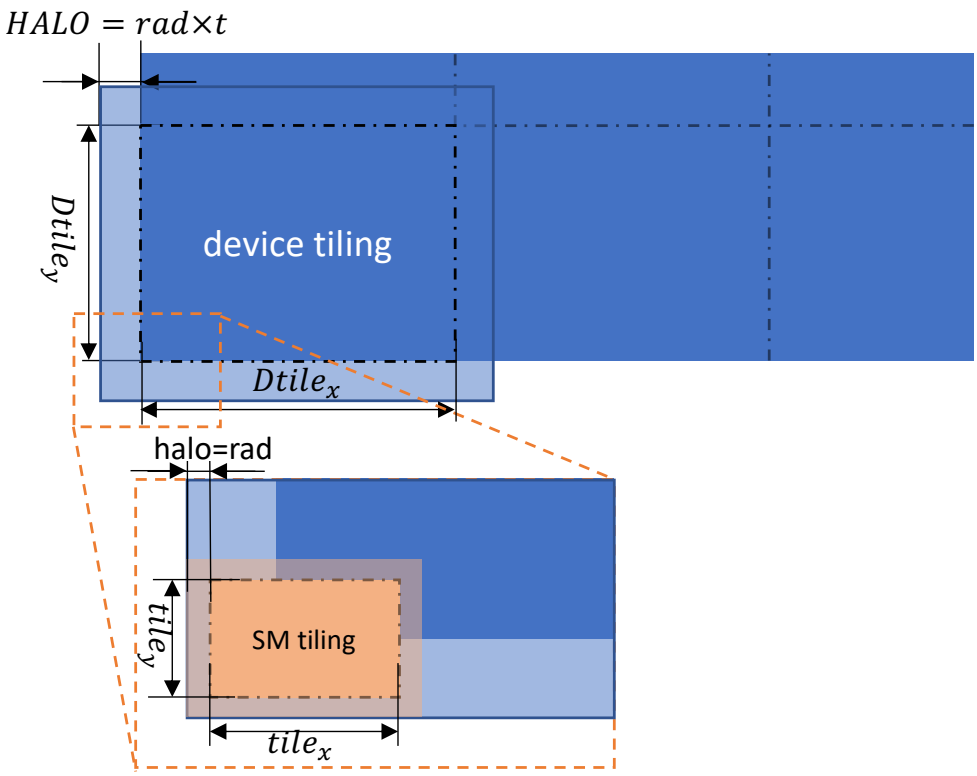


Figure 5.6: 2D Spatial tiling at the GPU device level.

or shared memory to reduce the resource burden.

5.5. Device Tiling: Reshaping Temporal Blocking with PERKS

The execution model PERKS presents a novel perspective to restructure existing optimization techniques. In this section, we apply the principles of PERKS to enhance temporal blocking optimization, which we refer to as device tiling.

In device tiling, we tile the domain such that a single tile can scale up to use the entire on-chip memory capacity of the GPU. Next, we let the tile reside in the on-chip memory while updating the cells for a sufficient number of time steps to amortize the initial loading and final storing overheads. We then store the final result for the tile on the device, and then we move to the next tile, i.e., the entire GPU is dedicated to computing only one single tile at any given time. Figure 5.6 shows how we do spatial tiling at the device level. We assume $tile_x \times tile_y$ to be the thread block tile configuration and $Dtile_x \times Dtile_y$ to be the device tile configuration. Thus, $(tile_x + halo \cdot 2) \times (tile_y + halo \cdot 2)$ is the total on-chip memory consumed at the stream multiprocessor level. $(Dtile_x + HALO \cdot 2) \times (Dtile_y + HALO \cdot 2)$ is the total on-chip memory consumed at the device level, where $HALO = rad \cdot t$. Additionally, figure 5.1.c shows the dependency between thread blocks that we must resolve. We use the bulk synchronous parallel (BSP) model to exchange the halo region and CUDA's grid level barrier for synchronization. We transpose the halo region

Listing 5.8: Pseudocode for 2D 5-Point Jacobian stencil device level spatial tiling.

```
void __global__ void device_2d5pt(...) {...
    //data is loaded from on-chip memory ocm_in
    //store data to ocm_out
    //ocm range tile_x, and tile_y;
    for(int s=0; s<t; s++){
        for(int l_y=0; l_y<tile_y; l_y+=1){
            for(int l_x=0; l_x<tile_x; l_x+=blockDim.x){
                ocm_out[i][j]=    a*ocm_in[i-1][j]
                                 +b*ocm_in[i][j]
                                 +c*ocm_in[i+1][j]
                                 +d*ocm_in[i][j-1]
                                 +e*ocm_in[i][j+1];
            }
        }
        __syncthreads();
        push_halo_to_neighbor(ocm_out[][], global_memory);
        grid.sync();
        swap(ocm_out,ocm_in);
        pull_halo_from_neighbor(ocm_in[][], global_memory);
        __syncthreads();
    }
...
}
```

that originally did not coalesce to reduce the memory transactions. Note that device tiling is an additional layer on top of SM tiling. Figure 5.6 shows an example of 2D spatial tiling at the device level, and Listing 5.8 presents the pseudocode of a 2D 5-point Jacobian stencil with device-level spatial tiling.

As such, we propose two methodologies for implementing temporal blocking: ❶ device-wide overlapped tiling and ❷ multi-queue-based solution (discussed in Section 5.4.2).

In device-wide overlapped tiling, we move the time loop from the host side to inside the kernel. Then, we spatially tile the domain of the problem and sequentially execute the tiles. We complete each tile's time steps before starting the next one.

In a circular multi-queue-based solution, we replace the 1D circular points (domain cells) in Figure 5.4 with the device tiles. In the device tiling situation, the sync(); function should be replaced by device (grid) level synchronization

5.6. Practical Attainable Performance

In this section, we analyze the practical attainable performance of temporal blocking by incorporating an overhead analysis (we derive valid proportion $\mathbb{V}$ from overhead analysis in Section 5.6.2) to a roofline-like model [27, 28] that predicts the attainable performance ($\mathbb{P}$ in Section 5.6.1). We project the practical attainable performance $\mathbb{PP}$ as:

$$\mathbb{PP} = \mathbb{P} \times \mathbb{V} \tag{5.1}$$

The model proposed in this section serves as a guide for implementation design choices in Section 5.7.

5.6.1. Attainable Performance

We use the giga-cells updated per second (GCells/s) as the metric for stencil performance [5,8]. We consider three pressure points in a stencil kernel: double precision ALUs, cache bandwidth (i.e., shared memory bandwidth in this paper), and device memory bandwidth (GPU global memory in this paper). Note that registers could also be a pressure point in extreme cases of very high-order stencils (outside the scope of this paper).

Assuming that the global memory bandwidth is $\mathbb{B}_{gm}$, the shared memory bandwidth is $\mathbb{B}_{sm}$, and the compute speed is $\mathbb{THR}_{cmp}$, the total access time is $\mathbb{A}_{gm}$ and $\mathbb{A}_{sm}$ for global memory and shared memory, respectively. The total amount of computation is $\mathbb{A}_{cmp}$. The memory access time per cell is a_{gm} and a_{sm} for global memory and shared memory, respectively; flops per cell is a_{cmp}. The total number of cells in the domain of interest is $\mathbb{D}_{gm}$, $\mathbb{D}_{sm}$ and $\mathbb{D}_{gm}$ for global memory, shared memory, and computation, respectively. The size of the cell (in Bytes) per cell is $\mathbb{S}_{Cell}$. We can compute the runtime of using each component to be:

$$\mathbb{T}_{gm} = \frac{\mathbb{A}_{gm}}{\mathbb{B}_{gm}} \times \mathbb{S}_{Cell} = \frac{a_{gm} \times \mathbb{D}_{gm}}{\mathbb{B}_{gm}} \times \mathbb{S}_{Cell} \tag{5.2}$$

$$\mathbb{T}_{sm} = \frac{\mathbb{A}_{sm} \times t}{\mathbb{B}_{sm}} \times \mathbb{S}_{Cell} = \frac{a_{sm} \times \mathbb{D}_{sm} \times t}{\mathbb{B}_{sm}} \times \mathbb{S}_{Cell} \tag{5.3}$$

$$\mathbb{T}_{com} = \frac{\mathbb{A}_{cmp} \times t}{\mathbb{THR}_{cmp}} = \frac{a_{cmp} \times \mathbb{D}_{cmp} \times t}{\mathbb{THR}_{cmp}} \tag{5.4}$$

The total runtime of the stencil is projected as:

$$\mathbb{T}_{stencil} = \max(\mathbb{T}_{gm}, \mathbb{T}_{sm}, \mathbb{T}_{cmp}) \tag{5.5}$$

The component c is the bottleneck if it satisfies:

$$\mathbb{T}_{c} = \mathbb{T}_{stencil} \tag{5.6}$$

We project the attainable performance $\mathbb{P}$ as:

$$\mathbb{P} = \frac{\mathbb{D}_{all} \times t}{\mathbb{T}_{stencil}} \tag{5.7}$$

Normally, we consider $\mathbb{D}_{all} = \mathbb{D}_{sm} = \mathbb{D}_{gm} = \mathbb{D}_{cmp}$. However, this is a case-by-case factor that depends on the implementation, e.g., when applying *device tiling* $\mathbb{D}_{gm} \neq \mathbb{D}_{sm}$.

5.6.2. Overheads

In this section, we discuss the overheads of different spatial blocking methods used in this paper:

Table 5.1: Design choices for EBISU.

Type	2D stencils	3D stencils
Parallelism Combination($TLP \times ILP$)	256×4	256×4
SM Tiling ($tile_x \times tile_y$)	256×4	32×32
Device Tiling	–	(12×6)
Temporal Blocking Strategy	Deep enough to shift the bottleneck	As deep as possible
Circular Multi-Queue	Compute	Shifting

Table 5.2: A100-PCIE Specifications.

	Specifications
DFLOPS	9.7 TFLOPS/s
Shared Memory Capacity	164 KB/ Stream Multiprocessor
Shared Memory Bandwidth	19.49 TB/s
Global Memory Bandwidth	1.555 TB/s

SM Tiling

The main overhead of SM tiling is related to redundant computation in the halo region (also known as the ghost zone [29]). Only a portion of the computation is valid. This valid portion is related to both the spatial and temporal block sizes and the radius of the stencil. In 2D stencils, we have:

$$\mathbb{V} = \frac{tile_x - 2 \times t \times rad}{tile_x} \tag{5.8}$$

In 3D stencils, we have:

$$\mathbb{V}_{SMtile} = \frac{(tile_x - 2 \times t \times rad) \times (tile_y - 2 \times t \times rad)}{tile_x \times tile_y} \tag{5.9}$$

Accordingly, we have:

$$\mathbb{PP}_{SMtile} = \mathbb{V}_{SMtile} \times \mathbb{P} \tag{5.10}$$

Device Tiling

The main overhead of the device level tiling is related to the overhead of synchronization. Only a portion of the runtime is valid. The valid portion depends on the runtime of the stencil ($\mathbb{T}_{stencil}$), the time required for device level synchronization ($\mathbb{T}_{Dsync}$) and the number of synchronization times per tile n (applying *lazy streaming* (Section 5.4) reduces n to 1):

$$\mathbb{V}_{Dtile} = \frac{\mathbb{T}_{stencil}}{\mathbb{T}_{stencil} + \mathbb{T}_{Dsync} \times n} \tag{5.11}$$

Accordingly, we have:

$$\mathbb{PP}_{Dtile} = \mathbb{V}_{Dtile} \times \mathbb{P} \tag{5.12}$$

To quantify the overhead, we followed the research of Zhang et al. [106] to test the overheads. The device (grid) level synchronization overhead in A100 is $\mathbb{T}_{Dsync} = 1.2us$.

5.7. EBISU: Analysis of Design Choices

In this section, we provide a comprehensive analysis to justify our design choices. The analysis is tailored for the A100 GPU (we summarize the parameters in Table 5.2), while it can be generalized to accommodate any GPU platform by adjusting the model parameters (Table 5.1 summarizes our findings on design choices).

We use 2D 5-Point (Listing 5.2) to represent 2D stencils, and 3D 7-Point (Listing 5.3) to represent 3D stencils for the discussions in this section. Table 4.2 shows the detailed parameters of both stencils.

5.7.1. Minimum Necessary Parallelism

As Section 2.4 showed, parallelism $\mathbb{PAR}$ is determined by thread-level parallelism TLP and instruction-level parallelism ILP. As such, to maintain a certain level of parallelism, we can reduce the occupancy (or TLP) and increase ILP simultaneously. We reduce the occupancy to the point that it will not increase the resources per thread block. In the current generation of GPUs (A100), reducing the occupancy of memory-bound kernels to less than 12.5% will not increase the available register per thread [64]. So, we set our aim conservatively at $Occupancy = 12.5\%$ or $TLP = 256$.

This research focuses on global memory access, shared memory access, and double precision FMA (Fused Multiply–Add),— fundamental operations in the stencil computation that we have examined. Our experimentation suggests that $ILP = 4$ and $Occupancy = 12.5\%$ ($TLP = 256$) provide enough parallelism for all three operations. We set this as a basic parallelism combination for our implementation. Note that the numbers above may vary for other GPUs, yet the analysis still holds.

5.7.2. Desired Depth

We use the attainable performance analysis (Section 5.6.1) to infer the desired depth. We aim at determining a sufficiently deep temporal blocking size to shift the bottleneck.

In this study, we are less concerned with whether the bottleneck shifts to computation or cache bandwidth. To simplify the discussion, we assume that the optimization goal is shifting the bottleneck from global memory to shared memory. This assumption is true for most of the star-shaped stencils [5]. Accordingly, we have the following:

$$\frac{a_{sm} \times t}{\mathbb{B}_{sm}} \times \mathbb{D}_{sm} \geq \frac{a_{gm}}{\mathbb{B}_{gm}} \times \mathbb{D}_{gm} \tag{5.13}$$

Case Study: 2D 5-Point Jacobian Stencil (representing stencils without device tiling)

Ideally, we have $\mathbb{D}_{sm} = \mathbb{D}_{gm}$. In A100, $\mathbb{B}_{gm} = 1555$ GB/s, $\mathbb{B}_{sm} = 19.49$ TB/s. In our 2D 5-point implementation, $\mathbf{a}_{gm} = 2$ (assuming perfect caching), $\mathbf{a}_{sm} = 4$. According to Equation 5.13, we have $t \geq \frac{a_{gm}}{a_{sm}} \times \frac{\mathbb{B}_{sm}}{\mathbb{B}_{gm}} = \frac{2 \times 19.49}{4 \times 1.555} = 6.3$. In $t = 7$, we measured the performance of 440 GCells/s. We can fine-tune to achieve slightly better performance at $t = 12$, where we measured 482 GCells/s.

There is only a 10% difference in achieved performance. The slight inaccuracy might come from the fact that, on average, the global memory accesses per data point are not perfectly cached.

Case Study: 3D 7-Point Jacobian Stencil (representing stencils with device tiling)

In device tiling 3D 7-point stencil, $\mathbb{D}_{gm}$ must also include the halo region between thread blocks. As such, we have:

$$\mathbb{D}_{gm} = (tile_x \times tile_y) + (tile_x + tile_y) \times 2 \times t \times rad \tag{5.14}$$

We intend to determine a t that satisfies the following:

$$\frac{a_{sm} \times \mathbb{D}_{sm} \times t}{\mathbb{B}_{sm}} \geq \frac{a_{gm} \times \mathbb{D}_{gm}}{\mathbb{B}_{gm}} \tag{5.15}$$

In the 3D situation, we have the following:

$$\frac{a_{sm} \times tile_x \times tile_y \times t}{\mathbb{B}_{sm}} \geq \frac{a_{gm} \times (tile_x \times tile_y \times 2 + (tile_x + tile_y) \times 2 \times t)}{\mathbb{B}_{gm}} \tag{5.16}$$

We assume that the following:

$$\mathbb{B}_{gm} \times a_{sm} \times tile_x \times tile_y - \mathbb{B}_{sm} \times a_{gm} \times 2 \times (tile_x + tile_y) > 0 \tag{5.17}$$

We have the following:

$$t \geq \frac{\mathbb{B}_{sm} \times a_{gm} \times tile_x \times tile_y \times 2}{\mathbb{B}_{gm} \times a_{sm} \times tile_x \times tile_y - \mathbb{B}_{sm} \times a_{gm} \times 2 \times (tile_x + tile_y)} \tag{5.18}$$

We assume that $tile_x = tile_y = 32$. We have the following:

$$t \geq \frac{\mathbb{B}_{sm} \times a_{gm} \times tile_x \times 2}{\mathbb{B}_{gm} \times a_{sm} \times tile_x - \mathbb{B}_{sm} \times a_{gm} \times 4} \tag{5.19}$$

Additionally, we have $\mathbf{a}_{sm} = 4.5$, $\mathbf{a}_{gm} = 2$. So we can get $t >= \frac{19.49 \times 2 \times 2 \times 32^2}{1.555 \times 4.5 \times 32^2 - 19.49 \times 4.5 \times 4 \times 32} = 18.34$. In this situation, the on-chip memory per thread block desired for EBISU is 352 KB, which exceeds the capacity of A100 (164 KB).

5.7.3. Device Tiling or SM Tiling?

Device tiling trades redundant computation for device level synchronization. In this section, we focus on: in EBISU, the performance implications of using one single tile per device (w/ device level synchronization). By comparing the practical attainable performance with the version that is not using one single tile per device (w/o device level synchronization).

Case Study: 2D 5-Point Jacobian Stencil

In 2d5pt, we have $\mathbb{T}_{stencil} = \mathbb{T}_{sm}$ for the overlapped tilling and the device level tiling. We simplify the discussion by defining a valid proportion $\mathbb{V}$, i.e., the updated output after excluding the halo. The higher the valid proportion, the higher the performance $\mathbb{P}$.

In overlapped tiling, for 2d5pt we have $t = 7$ (Section 5.7.2) and $rad = 1$. So $\mathbb{V}_{SMtile} \approx 95\%$

For device level tiling, we can go as deep as $t = 15$. So, we have: $\mathbb{T}_{sm} = \frac{t \times a_{sm} \times tile_x \times tile_y}{\mathbb{B}_{sm}} = 3.40$ us. Because $\mathbb{T}_{Dsync} = 1.2$ us. Accordingly, we have $\mathbb{V}_{Dtile} = \mathbb{T}_{sm}/(\mathbb{T}_{sm} + \mathbb{T}_{Dsync}) \approx 73\%$.

So, we have: $\mathbb{V}_{Dtile} \ll \mathbb{V}_{SMtile}$.

For 2D stencils of other shapes, we get:

$$\mathbb{PP}_{Dtile}(2D) \ll \mathbb{PP}_{SMtile}(2D) \tag{5.20}$$

In ideal scenery, we don't need on-chip resources for buffering. In A100, we have 164-kilo bytes or 21504 double precision data cells. This means ideally $\mathbb{T}_{sm} = 3.72us$, thus $\mathbb{V}_{Dtile} = 75\%$

As a result, in 2D stencils, the overhead of thread block level overlapped tiling is negligible, making device tiling less beneficial. This result stands true for all 2D stencils we studied in A100.

Case Study: 3D 7-Point Jacobian Stencils

In 3d7pt, we cannot shift the bottleneck to shared memory in overlapped (within acceptable overhead) or device tiling. We need to compare the Practical Attainable Performance in both cases to judge.

We have $\mathbb{V}_{SMtile} = (34 - 2 \times rad \times t)^2/34^2$. In 3d7pt, we have $rad = 1$, $t = 3$, $\mathbb{V}_{SMtile} \approx 77\%$. In $t = 3$, we have $\mathbb{P}_{SMtile} = 292$ GCells/s, and $\mathbb{PP}_{SMtile} \approx 225$ GCells/s.

On the other hand, for device tiling, we can go as deep as $t = 8$, so we have $\mathbb{L}(gm) = 2.42$. Because $\mathbb{T}_{Dsync} = 1.2$ us. So, $\mathbb{V}_{Dtile} \approx 67\%$ GCells/s. In $t = 8$ we have $\mathbb{P}_{Dtile} = 365$ GCells/s. Accordingly, we have $\mathbb{PP}_{Dtile} \approx 244$ GCells/s.

So, we have: $\mathbb{PP}_{Dtile} > \mathbb{PP}_{SMtile}$ on a 3d7pt stencil.

We measured, for instance, 151 GCells/s for w/o device tiling and 197 GCells/s for with device tiling. The experiment results are consistent with the analysis (for 3D stencils of other shapes as well):

$$\mathbb{PP}_{Dtile}(3D) > \mathbb{PP}_{SMtile}(3D) \tag{5.21}$$

As a result, for 3D stencils, the overhead of thread block level overlapped tiling is so significant that it prohibits the temporal blocking implementation from going deeper. This result stands true for all 3D stencils we studied in A100.

Based on the analyses above, we only implement device tiling for 3D stencils in EBISU. The analysis in the following section is built on top of this decision.

5.7.4. Deeper or Wider?

As the capacity of on-chip memory is limited, there is a trade-off between increasing the width of spatial blocking and increasing the depth of temporal blocking. This section discusses the heuristic we use for parameter selection in EBISU.

Case Study: 2D 5-Point Jacobian Stencil

Firstly, as Section 5.7.3 showed, the overhead of 2D 5-Point Jacobian Stencil is negligible. Additionally, according to Section 5.7.2, in theory, at depth $t = 7$, we shift the bottleneck from global memory to shared memory.

As such, after the bottleneck is shifted, we aim at wider spatial blocking to reduce the overhead of overlapped tilling as is discussed in Section 5.6.2. Yet, we still need to consider the implementation simplicity. For example, we choose a tiling of size $tile_x = 256$ instead of $tile_x = 328$, since the latter is hard to implement in CUDA. .

Case Study: 3D 7-Point Jacobian Stencil

For simplicity, we assume that the very first plane loaded and the last plane stored have already been amortized. Then, for global memory access, we only focus on the halo region. According to Equation 5.13, we have:

$$\frac{tile_x \times tile_y \times a_{sm}}{\mathbb{B}_{sm}} > \frac{(tile_x + tile_y) \times 2 \times a_{gm} \times rad}{\mathbb{B}_{gm}} \tag{5.22}$$

We assume that $tile_y = tile_x$. So, we can get the following:

$$tile_y = tile_x > \frac{4 \times a_{gm} \times \mathbb{B}_{sm}}{a_{sm} \times \mathbb{B}_{gm}} \times rad \tag{5.23}$$

In our 3d7pt implementation, $a_{gm} = 2$, $a_{sm} = 4.5$. We have $tile_y = tile_x \geq 22.3$. For implementation convenience, we use 32×32 (also fitted to the minimal necessary parallelism that saturates the device as Section 5.7.1 discussed). As such, after the spatial tiling is large enough for overlapping halo regions, we run the temporal blocking as deep as possible to amortize the overhead of using device (grid) level synchronization.

5.8. Evaluation

We experiment on various 2D and 3D stencils (listed in Table 5.3). The test data are generated by STENCILGEN [4]. We evaluate the benchmarks on an NVIDIA A100-PCIe GPU device(host CPU: Intel Xeon E5-2650).

5.8.1. Compile Settings of EBISU

The code is compiled with NVCC-11.5 (CUDA driver V11.5.119) and gcc-4.8.5, using flags -rdc=true -Xptxas "-v " -std=c++14. We only generate code for A100 architecture [1]. The "-rdc=true" flag is necessary to enable grid level synchronization, so we set it by default. We use c++14 features, so we add "-std=c++14" flag. -Xptxas "-v" is set to gather information on registers.

[1] setting CUDA_ARCHITECTURES "80" in CMAKE

Table 5.3: Stencil benchmarks. Readers can refers to [4,5] for a details description. We also include ideal shared memory access times per cell in this research, a_{sm}, when applying redundant register streaming (w/ RST) and without it (w/o RST) in the table.

Stencil **[Order, FLOPs/Cell]**	**Domain Size**	a_{sm} **w/o RST**	a_{sm} **w/ RST**
j2d5pt [1,10]	8352^2	6	4
j2d9pt [2,18]	8064^2	10	6
j2d9pt-gol [1,18]	8784^2	10	4
j2d25pt (gaussian) [2,25]	8640^2	26	6
j3d7pt (heat) [1,14]	$2560 \times 288 \times 384$	8	4.5
j3d13pt [2,26]	$2560 \times 288 \times 384$	14	7
j3d17pt [1,34]	$2560 \times 288 \times 384$	18	5.5
j3d27pt [1,54]	$2560 \times 288 \times 384$	28	5.5
poisson [1,38]	$2560 \times 288 \times 384$	20	5.5

Table 5.4: Depth of temporal blocking for each stencil implementations in this evaluation.

Type	**STENCILGEN**	**AN5D**	**DRSTENCIL**	**ARTEMIS**	**EBISU**
j2d5pt	4	10	3	12	12
j2d9pt	4	5	2	6	8
j2d9pt-gol	4	7	2	6	6
j2d25pt	2	5	2	3	4
j3d7pt	4	6	3	3	8
j3d13pt	2	4	2	1	5
j3d17pt	2	3	2	2	6
j3d27pt	2	3	-	2	5
poisson	4	3	2	2	6

5.8.2. Evaluation Setup

Domain Size

We used the domain sizes listed in Table 5.3 for EBISU, comparable to those used in the literature [8, 54, 139].

Warm-Up and Timing

We do warm-up iterations for all experiments and then use GPU event APIs to measure one kernel run. We repeat this process ten times and report the peak.

Depth of Temporal Blocking

We only evaluate a single kernel. Therefore, the total number of time steps equals the depth of temporal blocking of each implementation in each stencil benchmark. Table 5.4 summarizes the depth of temporal blocking.

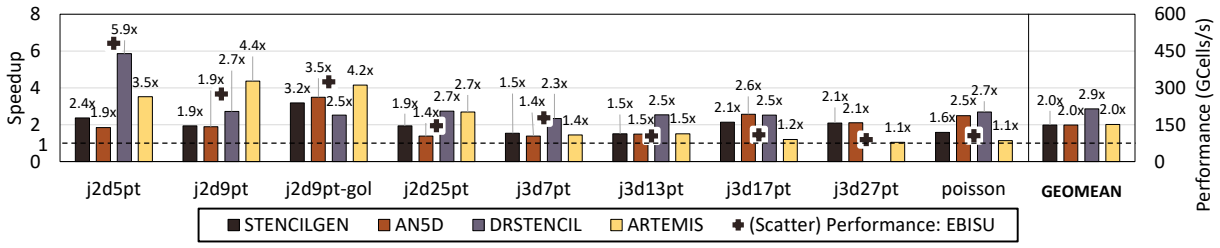


Figure 5.7: Speedup of EBISU over the state-of-the-art temporal blocking implementations. We also plot the performance of EBISU (right Y-axis plotted as '+' ticks).

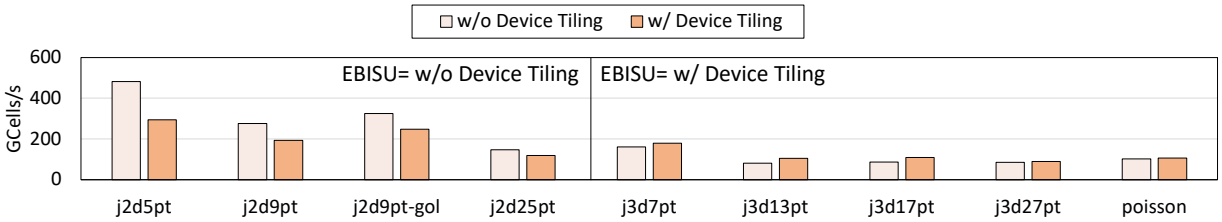


Figure 5.8: The performance of using applying device tiling versus not applying device tiling. EBISU solely applies device tiling in 3d stencils.

5.8.3. Comparing with State-Of-The-Art Implementations

We compare EBISU with the state-of-the-art temporal blocking implementations AN5D [5] and STENCILGEN [4], and the state-of-the-art auto-tuning tools ARTEMIS [137] and DRSTENCIL [140].

Setting up State-Of-The-Art Libraries

We use the domain sizes reported by each library in the original paper (not adversely changing domain sizes). We assume that the libraries can achieve reasonably good performance in the setting used in the original paper. For example, in 2D stencils, AN5D used 16384^2, while STENCILGEN used 8192^2. ARTEMIS did not report 2D stencils; we used the same setting as STENCILGEN. Details can be obtained from the original papers [4, 5, 137, 140].

As for timing and warm-up. AN5D's original code already does the warm-up, so we use the default setting. We use the same host warm-up and timer function as EBISU to test the kernel performance for STENCILGEN, ARTEMIS, and DRStencil.

The detailed settings are listed as follows:

STENCILGEN We used the codes for AD/AE appendix [2] of the original paper. We do not change anything inside the kernel.

AN5D AN5D is a code auto-generator tool. We only used the code already generated in their

[2]https://github.com/pssrawat/IEEE2017

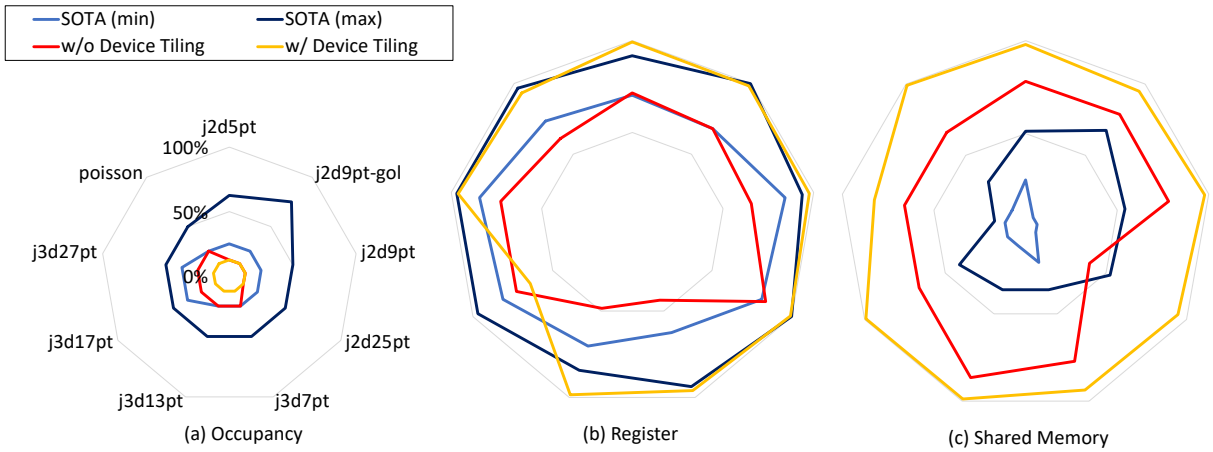


Figure 5.9: Percent of occupancy achieved and resources used (registers and shared memory) for EBISU, both with and without device tiling, s well as for state-of-the-art libraries across all stencil benchmarks.

code [3]. We port the makefile system to A100 and iterate over all generated codes to find the highest performance for each stencil benchmark. The original code did not include some stencil benchmarks we use. We use the implementations with similar memory access patterns to represent them: gaussian (box2d2r), j3d7pt (star3d1r), j3d13pt (star3d2r), j3d17pt (j3d27pt) and poisson (j3d27pt).

DRSTENCIL DRSTENCIL [140] is also an auto-tuning tool. We use the benchmark in the codebase [4]. In the paper, the authors included only the implementations of the j3d7pt stencil in the range of 3D stencils. We extend their j3d7pt stencil setting to other 3D stencils for comparison. However, with the j3d7pt setting, DRSTENCIL could not generate executable code in j3d27pt. We report the kernel with the peak performance among the policies that DRSTENCIL iterated over.

ARTEMIS ARTEMIS is an auto-tuning tool. We use the benchmark in the codebase [5]. We replaced the profiler nvprof (deprecated) with ncu. ARTEMIS [137] only provides samples for 3d7pt and 3d27pt. We extend 3d7pt to all star-shape stencils (including heat and 2d star-shape stencils) and 3d27pt to all box-shape stencils (including poisson, 3d17pt and 2d box-shape stencils). We report the kernel with the peak performance among the policies that ARTEMIS iterated over.

Performance Comparison

Figure 5.7 shows the speedup of EBISU over state-of-the-art temporal blocking implementations. EBISU shows a clear performance advantage over all of the state-of-the-art temporal blocking libraries, i.e., STENCILGEN and AN5D. It is also faster than the state-of-the-art auto-tuning tool DRSTENCIL and ARTEMIS. EBISU achieves a geomean speedup of over 2.0x when comparing

[3] https://github.com/khaki3/AN5D-Artifact
[4] https://github.com/simple86/DRStencil
[5] https://github.com/pssrawat/artemis

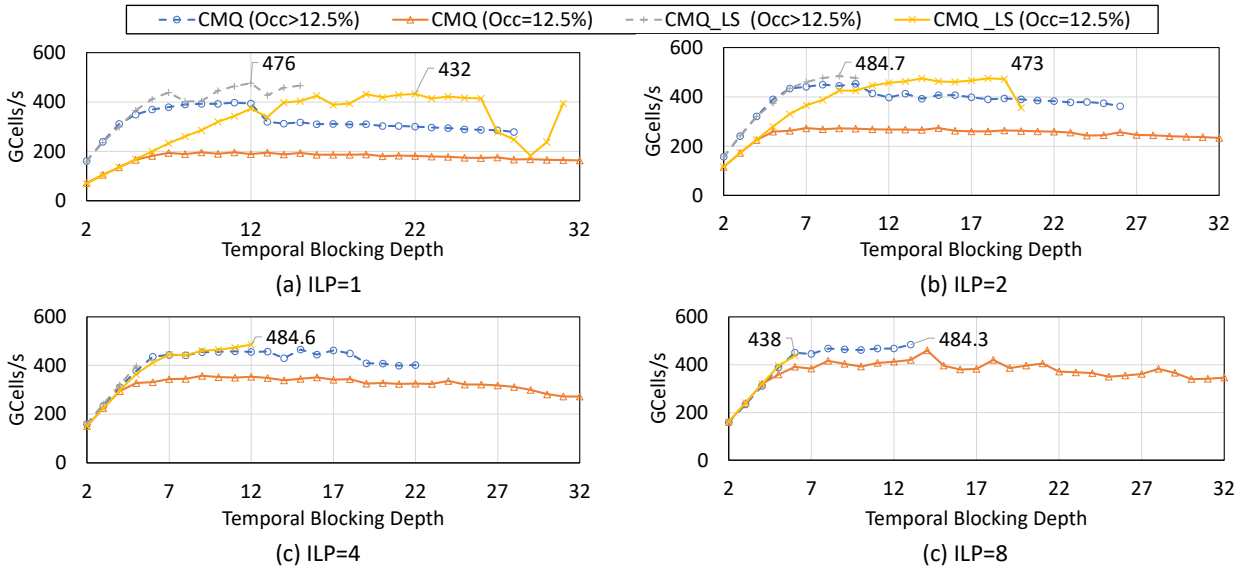


Figure 5.10: The setting of different combinations of policies in EBISU, i.e., circular multi-queue (CMQ), lazy streaming (LZ), occupancy (Occ), and the depth of temporal blocking, and its corresponding performance (GCells/s).).

with each state-of-the-art. When comparing EBISU with the best state-of-the-art in each stencil, EBISU achieves a geomean speedup of 1.49x.

5.8.4. Device Tiling or SM Tiling?

Experiments were conducted to ascertain the optimal usage scenarios for device tiling, with the results presented in Figure 5.8. Evidently, SM tiling outperforms device tiling in 2D stencils, whereas the latter exhibits superior performance in 3D stencils. This is because in 2d stencils overlapped tiling has lower overhead. Conversely, the high overhead associated with overlapped tiling in 3D stencils makes the inclusion of device synchronization and the application of device tiling, a profitable trade-off. This result alins with the analysis conducted in Section 5.7.3.

5.8.5. Parameter Study of 2D 5-Point Jacobian stencils

The impact of various factors within EBISU can be challenging to isolate. For instance, choosing not to implement lazy streaming optimization might free up additional resources that could be utilized for higher occupancy or deeper temporal blocking. In this section, we aim to examine the influence of several factors: the use or non-use of lazy streaming, level of occupancy, level of instruction-level parallelism, and the depth of temporal blocking. We present case studies involving 2D 5-Point Jacobian stencils (representing 2D stencils). The results are depicted in Figure 5.10, leading to the following observations:

Admittedly, higher parallelism generally yields better performance at the same depth. Yet, increased parallelism consumes more resources, limiting EBISU's capability to attain deeper tem-

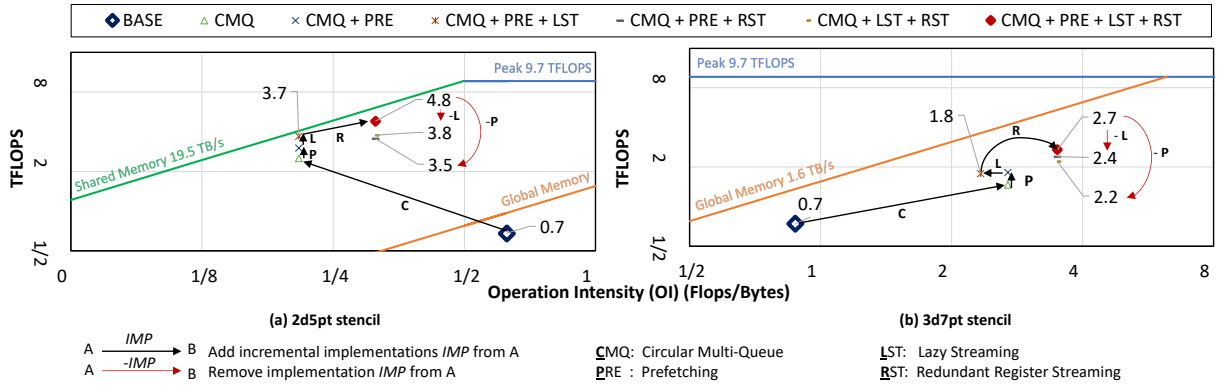


Figure 5.11: Roofline plots for different implementations in Section 5.4. We plot 2D 5-Point Jacobian stencil implementations to represent 2D stencils and 3D 7-Point Jacobian stencil implementations to represent 3D stencils. The black arrows link the incremental implementations from a *BASE* implementations. The 3D *BASE* applies device tiling (Section 5.5). The *LST* refers to thread block level lazy streaming. Device tiling without lazy streaming will be extremely slow as can be inferred in Section 5.6.2.

poral blocking. Tuning parameters related to parallelisms, such as occupancy and instruction-level parallelism, can marginally enhance performance (from 484.6 GCells/s to 484.7 GCells/s, improving 0.2%), validating our assumption that once the device reaches saturation with minimal parallelism, additional parallelism negligibly affects performance. Further increments in parallelism do not significantly impact performance more than good resource scaling schemes.

Resources

We also present data on occupancy and resource utilization for all benchmarks, obtained via the ncu profiler (see Figure 5.9). Generally, all EBISU implementations, regardless of whether to apply device tiling, can use the on-chip resources efficiently, despite exhibiting low occupancy.

EBISU without device tiling version utilizes fewer registers than its equivalent state-of-the-art kernels. Worth noting that, in 3d scenario, due to the overhead of redundant computation, the temporal blocking depth is relatively shallow such that it is allowed to have a relatively higher occupancy compared with the device tiling counterpart.

Conversely, the EBISU device tiling version heuristically consumes as many resources as feasible, thereby exhibiting minimal occupancy but nearly always consuming more on-chip resources, namely registers and shared memory, compared to all other implementations.

5.8.6. Performance Breakdown

The remarkable speedup EBISU achieved compared to other SOTA methods can be attributed to a fundamental shift in GPU programming principles. While existing SOTAs typically focus on constraining resources to enhance parallelism, EBISU constrains parallelism to optimize resource

utilization. This novel approach enables the implementation of resource-scalable schemes, ultimately contributing to EBISU's performance.

In this section, we provide a detailed explanation of how the optimizations proposed in earlier sections impact the performance of EBISU. To demystify their effects, we present case studies involving 2D 5-Point Jacobian stencils (representing 2D stencils) and 3D 7-Point Jacobian stencils (representing 3D stencils). Figure 5.11 displays the roofline plot of various implementations, with the black arrow indicating the incremental implementation of each scheme.

For the roofline analysis, we report the performance as measured in TFLOPS (teraflops). Table 4.2 shows the relationship between TFLOPS and GCells/s metrics.

BASE

The BASE implementation refers to the approach that applies minimal parallelism analysis, as discussed in Section 5.7.1. In this phase, we prepare the necessary resources for EBISU. It is important to note that in the case of the 3D 7-Point stencil, the BASE implementation already incorporates *device tiling*, similar to the approach employed in the existing research of PERKS [141].

Circular Multi-Queue (CMQ)

CMQ is a foundation for deep temporal blocking. As Figure 5.11 shows, in 2D stencils, we increase the depth of temporal blocking to move the bottleneck from global memory to shared memory. In 3D stencils, due to the shared memory's limited capacity, we only move the Operation Intensity (OI) from left to right. Either way, we move the OI to increase the attainable performance shown in the roofline model.

Prefetching (PRE)

As Figure 5.11 shows, the PRE scheme moves the roofline plot toward the attainable bound. However, it does not directly impact the attainable bound itself.

Lazy Streaming (LST)

The LST scheme aims to reduce synchronizations by using long buffers. By default, we employ LST to minimize device level synchronizations. This section specifically focuses on the impact of LST on reducing thread block synchronizations. As illustrated in Figure 5.11.a, applying LST to the 2D 5-point stencil brings its performance closer to the attainable bound. However, in the case of the 3D stencil, as shown in Figure 5.11.b, applying LST may harm performance. This is primarily due to the global memory still being the bottleneck, and the additional on-chip memory space required by LST implementation leads to a shallower temporal blocking. This results in a leftward shift in the OI, reducing the attainable performance. It is worth noting that in the final version of EBISU, disabling LST for the 3D 7-point stencil allows for a doubling of the temporal blocking depth, from $t = 8$ to $t = 16$, leading to a performance increase from 2.7 TB/s to 2.9 TB/s. However, when excluding the redundant halo, the performance dips from 2.4 TB/s to 2.3 TB/s. Therefore, this

result has been excluded from the discussion

Redundant Register Streaming (RST)

RST's primary goal is to cut down shared memory access time (refer to Table 4.2). By doing so, we can shift the roofline plot closer to the compute-bound from left to right when shared memory is the bottleneck (as shown in Figure 5.11.a). Also, we leverage RST to cache a portion of the tiling, which helps reduce the amount of data cached in shared memory. This enables us to achieve deeper temporal blocking and move the roofline plots closer to the compute-bound from left to right when global memory remains the bottleneck (as shown in Figure 5.11.b).

Relations Between Optimizations

The PRE and LST optimizations improve performance and bring it closer to the attainable bound. The RST optimization is designed to shift the roofline plots to the right to increase the attainable bound. Red arrows in Figure 5.11 clearly show that disabling either of these optimizations results in performance degradation.

Practical Attainable Performance

In 2D 5-point stencil, we achieved 4.8 TFLOPS (80% of the attainable bound). In 3D 7-point stencil, we achieved 2.7 TFLOPS (50% of the attainable bound). The big gap is due to the omission of the overheads in the roofline model. As we consider overhead in our model (Section 5.6), we achieved 88% and 80% of $\mathbb{PP}$ in 2D 5-point and 3D 7-point stencils respectively. A model that considers the overheads can model the practical attainable performance better. As such, this model contributing to the decision-making also benefits the performance of EBISU.

Hit Rate

Figure 5.12 illustrate how applying EBISU influence the hit rate. We also include the PERKS statistics in the same figure for comparison. For simplicity, we use PERKS to represent device tiling (since they are equivalent). We summarize the observations as follows:

EBISU generally enhances the L2 cache hit rate compared with baseline and improves the L1 cache hit rate in 2d stencils. However, for 3d stencils, it inevitably reduces the L1 cache hit rate. This reduction partly explains why EBISU does not perform equally well when applied to 3D stencils as to 2D stencils.

5.9. Desired Scratchpad Memory Capacity

5.9.1. A Case Study on 2D 5-Point Jacobian Stencil

In this section, we analyze the impact of machine balance on the desired capacity for EBISU. We use a 2D 5-Point Jacobian Stencil as a case study. Because most machine specifications don't release the information of cache bandwidth, we use machine balance instead to determine the desired depth.

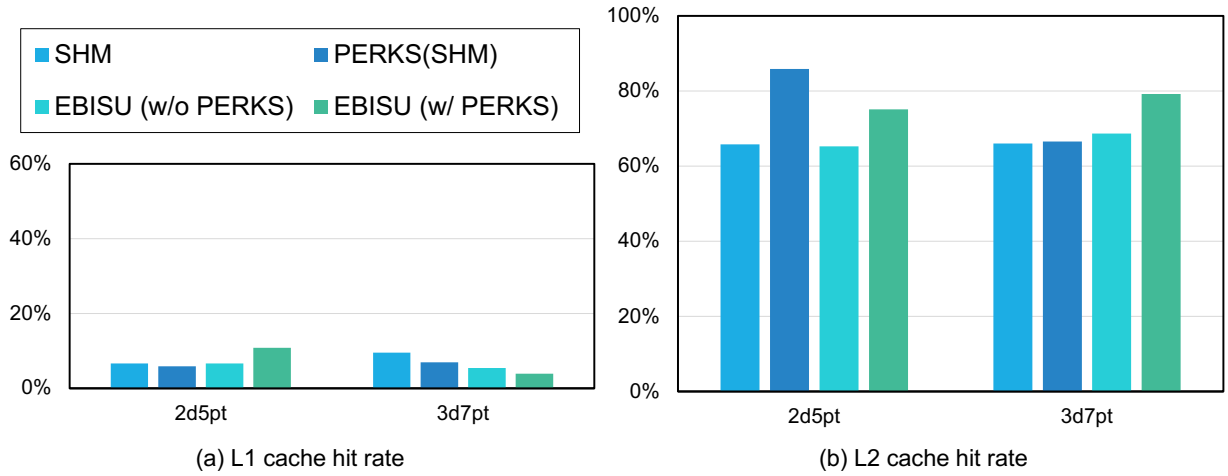


Figure 5.12: The hit rate of L1 cache and L2 cache when applying EBISU and PERKS. We use *EBISU (w/PERKS)* to represent *EBISU with device tiling* (since they are equivalent).

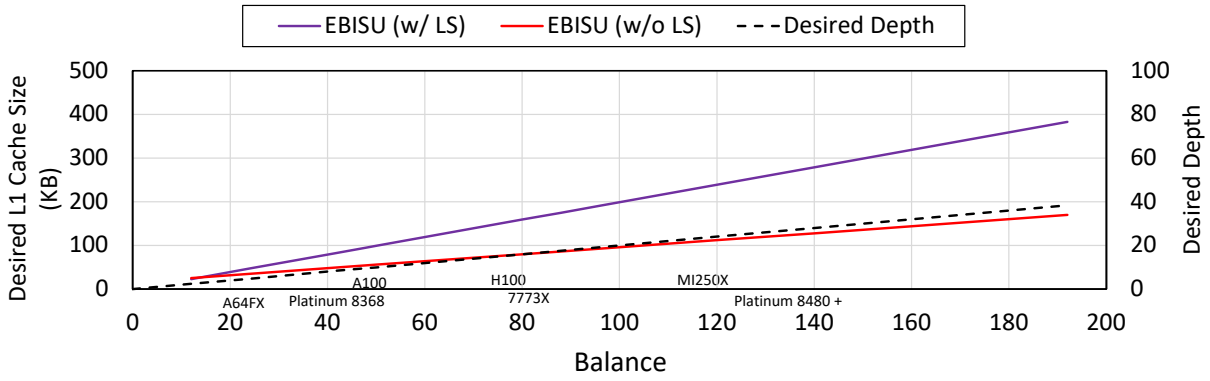


Figure 5.13: The desired cache capacity for EBISU (2d5pt stencil) across different machine balance.

Accordingly, we derive the following equation:

$$t \geq \frac{a_{gm}}{a_{cmp}} \times \frac{\mathbb{THR}_{cmp}}{\mathbb{B}_{gm}} \times \mathbb{C}_{Cell} = \frac{a_{gm}}{a_{cmp}} \times balance \tag{5.24}$$

Because operational intensity is given by $I = \frac{a_{cmp}}{a_{gm}} \times \mathbb{C}_{Cell}$, we also have:

$$t \geq \frac{balance}{I} \times \mathbb{C}_{Cell} \tag{5.25}$$

In the case of the 2D 5-Point Jacobian Stencil, we have $\frac{\mathbb{C}_{Cell}}{I} = \frac{a_{gm}}{a_{cmp}} = \frac{2}{10} = \frac{1}{5}$. Given the A100's balance of 49.9, we derive $t \geq 9.98$. This result is marginally deeper than the assumption that the scratchpad memory is the bottleneck. However, it gives a reasonable approximation of the desired depth.

With the desired depth, we can infer the desired cache capacity necessary to implement EBISU.

For situations that do not use lazy streaming, we have:

$$\mathbb{M}(EBISU_{w/oLazyStreaming}) = (tile_y + (rad \times 2 + tile_y) + rad \times t) \times tile_x \times \mathbb{S}_{Cell} \tag{5.26}$$

and for those that do use lazy streaming:

$$\mathbb{M}(EBISU_{LazyStreaming}) = (tile_y + (tile_y + rad) \times t) \times tile_x \times \mathbb{S}_{Cell} \tag{5.27}$$

We assume the same tiling settings as those used in this analysis. Thus, for a 2D 5-Point Jacobian Stencil with $tile_x = 256$, $tile_y = 4$, and $rad = 1$, we get:

$$\mathbb{M}(EBISU_{w/oLazyStreaming}) = (10 + t)KB \tag{5.28}$$

$$\mathbb{M}(EBISU_{LazyStreaming}) = (4 + 5 \times t)KB \tag{5.29}$$

Figure 5.13 illustrates the correlation between desired capacity and machine balance. We can observe that many imbalanced chips require a large L1 cache so that EBISU can fully function. So we consider extending EBISU to different cache hierarchies as a potential future work to address the challenges associated with imbalanced platforms equipped with a limited capacity of L1 cache. Further discussion on this matter will be presented in Section 6.3.3.

5.9.2. A Case Study on 3D 7-Point Jacobian Stencil

The desired tiling for a 3D 7-Point Jacobian Stencil (device tiling) can be similarly deduced:

$$tile_y = tile_x > \frac{4 \times a_{gm} \times \mathbb{THR}_{cmp}}{a_{cmp} \times \mathbb{B}_{gm}} \times \mathbb{C}_{Cell} \times rad == 4 \times rad \times \mathbb{C}_{Cell} \times \frac{balance}{I} \tag{5.30}$$

In the case of the 3D 7-Point Jacobian Stencil, we have $\frac{\mathbb{C}_{Cell}}{I} = \frac{a_{gm}}{a_{cmp}} = \frac{2}{14} = \frac{1}{7}$. Given the A100's balance of 49.9. Accordingly, we have $tile_x = tile_y = 28.51$. We utilize $tile_x = 32$ as the power of 2 is convenient to program in GPUs.

As for the desired depth, we have the following:

$$t \geq \frac{\mathbb{THR}_{cmp} \times a_{gm} \times tile_x \times 2}{\mathbb{B}_{gm} \times a_{cmp} \times tile_x / \mathbb{C}_{Cell} - \mathbb{THR}_{cmp} \times a_{gm} \times 4} = \frac{tile_x \times 2}{\frac{I}{balance \times \mathbb{C}_{Cell}} \times tile_x - 4} \tag{5.31}$$

Accordingly, for situations that do not use lazy streaming, we can infer that:

$$\mathbb{M}(EBISU_{w/oLazyStreaming}) = (tile_x + 2 \times rad)^2 \times ((1 + rad) \times t + 1) \times \mathbb{S}_{Cell} \tag{5.32}$$

And for those that do use lazy streaming, we have the following:

$$\mathbb{M}(EBISU_{LazyStreaming}) = (tile_x + 2 \times rad)^2 \times ((1 + 2 \times rad) \times t) \times \mathbb{S}_{Cell} \tag{5.33}$$

We aim to solve the following optimization problem to figure out the desired scratchpad memory capacity:

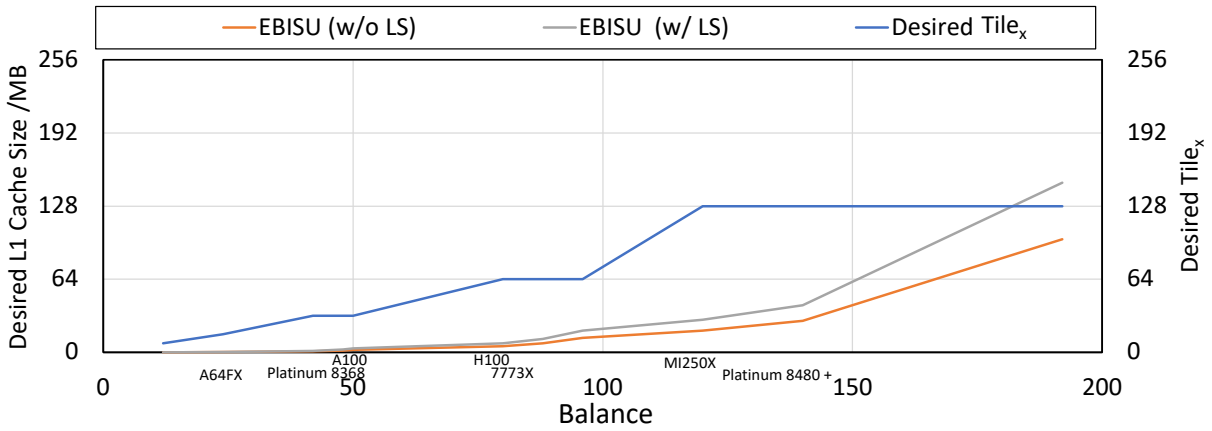


Figure 5.14: The desired cache capacity for EBISU (3d7pt stencil) across different machine balance.

$$\begin{aligned} \underset{balance,I}{\text{minimize}} \quad & \mathbb{M} \\ \text{subject to} \quad & t \geq \max(1, \frac{tile_x \times 2}{\frac{I}{balance \times \mathbb{C}_{Cell}} \times tile_x - 4}) \\ & tile_x > 4 \times rad \times \mathbb{C}_{Cell} \times \frac{balance}{I} \end{aligned} \tag{5.34}$$

Figure 5.14 illustrates the correlation between the desired L1 capacity and the machine balance. As depicted, a significantly large L1 cache capacity is required for 3d7pt stencils to shift the bottleneck. As such, in practical scenarios, aiming to maximize performance at a given capacity might be practical. We leave this parameter study for future work as we will discuss in Section 6.5.

5.10. Related works

Apart from the tiling optimizations we covered in Section 5.2.1, many stencil optimizations are architecture-specific. For example, vectorization [1, 142, 143]; cache optimizations on CPUs [46–49]. For GPUs [2, 72, 132], Chen et al. proposed an execution model on top of the shuffle operation on GPU [8]; Liu et al. uses tensor cores to accelerate low precision stencils [144]. Rawat et al. also summarized optimizations that can be used in stencil optimization, i.e., streaming, unrolling, prefetching [137], and register reorder [139].

State-of-the-art implementations are usually built on top of multiple optimizations. For example, wavefront diamond blocking [48] is built on top of vectorization, cache optimization, streaming, and diamond tiling, STENCILGEN [4] is built on top of shared memory optimization, streaming, and N.5D tiling. But combining different optimizations is tedious for implementation. Many pieces of research focus on auto code generation using domain-specific language [4, 129, 145], or compiler-based approaches [7, 146]. Some optimizations, especially those related to registers, are challenging to implement manually. Matsumura et al. implemented AN5D [5] that generates codes using registers effectively.

5.11. Conclusion

In this chapter, we propose EBISU, a novel temporal blocking approach. EBISU relies on low occupancy and mapping on large tiles over the device. The freed resources are then used to improve the data locality. We compared EBISU with two state-of-the-art temporal blocking implementations and two state-of-the-art autotuning tools. EBISU constantly shows its performance advantage. It achieves a geomean speedup of 1.49x over any of the top alternative state-of-the-art implementations for each stencil benchmark.

CHAPTER 6

Discussion and Future Work

This dissertation mainly addresses the challenge of the imbalance between compute and memory bandwidth. In this chapter, we discuss our research topics' insight and potential future research directions.

6.1. Parallelism

This thesis primarily explores occupancy as a representation of parallelism. One extension is to incorporate stream multiprocessors into consideration. Our research shows a lower occupancy is sufficient to saturate the memory bus, partly aligning with findings from previous studies, such as Darabi's 2022 research on Morpheus [42], which shows that saturation does not necessitate full utilization of all stream multiprocessors. This understanding introduces an additional realm of exploration for us. The notion of a minimized stream multiprocessor requirement is also extendable to CPUs. Hence, we can broaden the concept of on-chip resources to L2 cache within CPUs.

6.2. Generalization

6.2.1. Generalizing PERKS

We summarize the primary features of PERKS as: ❶ persistent kernel and ❷ use unused on-chip resources to cache data. Though persistent kernel is a concept in GPU programming, we anticipate the philosophy of PERKS can be generalized to any platform.

Take the CPU as an example. Most CPUs don't have scratchpad memory, posing a challenge to surpass the cache system and preserve a portion of the cache for long-term benefit. Additionally, CPUs handle context switching differently than GPUs: CPUs push the entire context into the stack rather than simply changing the program counter.

Alternatively, we anticipate a compiler and hardware support might ease the process of implementing PERKS. For example, the compiler might automatically determine that only 80% of registers will be consumed, leaving the 20% remaining resources available for PERKS use. Then, if the end-user determines the cacheable region, the compiler can automatically push a portion of the region to the reserved space.

Nevertheless, the generalization of PERKS is quite straightforward. We anticipate this feature will facilitate the integration of PERKS into productive scientific repositories, such as those used for earthquake simulations.

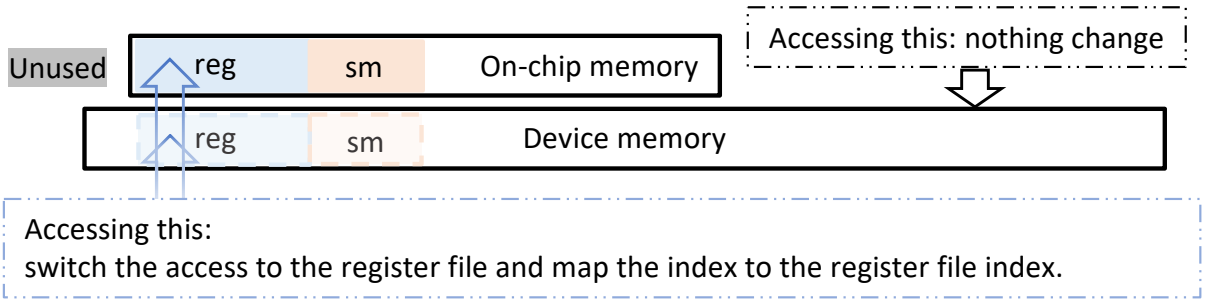


Figure 6.1: Private pointer mechanism

6.2.2. Generalizing EBISU

EBISU's essential features can be summarized as ❶ constraining parallelism and ❷ optimizing resource usage. As EBISU is a derivative of PERKS, its comprehensive extension is governed by the same limitations present in PERKS.

However, the philosophy behind EBISU, which constrain parallelism for performance and repurpose spared resources for code optimization, is independent of the underline architectures. It holds the potential to be universally applied across various architectures. We envisage this principle may also benefit a broad array of algorithms hampered by resource constraints.

In pursuit of this vision, we are actively investigating the applicability of this principle to an expanded range of computational tasks and applications, thereby strengthening EBISU's scope of influence.

6.3. On-chip Memory Support

6.3.1. Private Pointer Mapping

The concept of the private pointer is an extension idea from the PERKS execution model. Within PERKS, data dependencies still require management. However, if developers can ascertain that certain data sections will only be accessed within a specific thread or thread block, introducing thread-private or thread-block-private pointers can be beneficial.

Although these pointers reference global memory, they would enable the compiler to map sections of the pointed space to unused registers or scratchpad memory. Figure 6.1 illustrates an example of the private pointer mechanism.

6.3.2. Hardware and Compiler Support for Register Cache

In our current model, although we've successfully managed to use registers for caching, the inflexibility in accessing register files necessitates the continued use of shared memory as a temporary holder. This suggests that while using register files for caching reduces memory transactions to global memory, the transactions to shared memory may not experience a similar reduction.

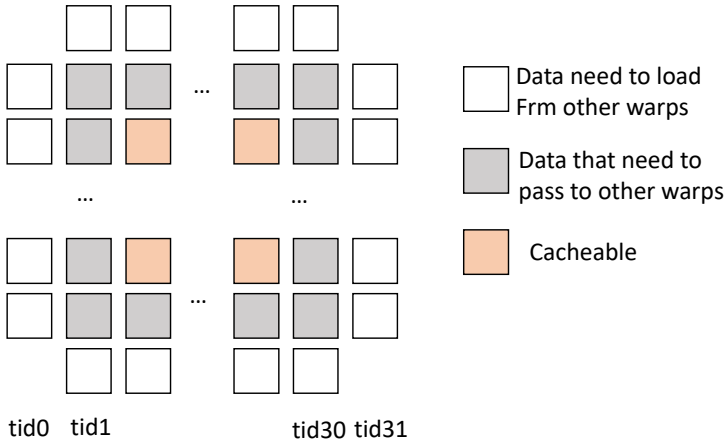


Figure 6.2: Cacheable region inside a tiling for a warp in stencil We load the maximum possible cacheable region (including halo) to compute T consecutive time steps. The halo region is required to resolve both temporal and inter thread block spatial dependency.

The use of registers, as elucidated below, has certain constraints:

- Within a warp, all threads perform exactly the same operations. Consequently, if one thread needs to access shared or global memory, every thread within the warp will mimic this operation.
- To leverage the register array, the index of the array must be statically determined during compile time.

Nonetheless, scenarios like stencil present unique data access patterns as depicted in Figure 6.2. The primary challenges here are:

- Tid 0 and tid 31 need to load their private array to scratchpad memory or global memory
- Tid 1 and tid 30 need to store their private array to scratchpad memory or global memory

Thus, even with caching in register files, the transactions to shared memory remain unchanged, and register files merely serve as an extension of shared memory.

Inspired by software systolic array [8], we envision a certain switched data access format as depicted in Figure 6.3. in this way, we can reduce shared memory access from 30 times to 12 times while using register file to cache data.

Nonetheless, this approach grapples with the issue that the index of the private array must be statically determined during compile time. With the current compiler, there's no way to use relative registers across a single warp or use warp id to determine register id (for example, reg 0 in thread 0, reg 1 in thread 1).

We anticipate that future compilers might support such alterations and thus PERKS could substantially benefit from merging with SSAS [8].

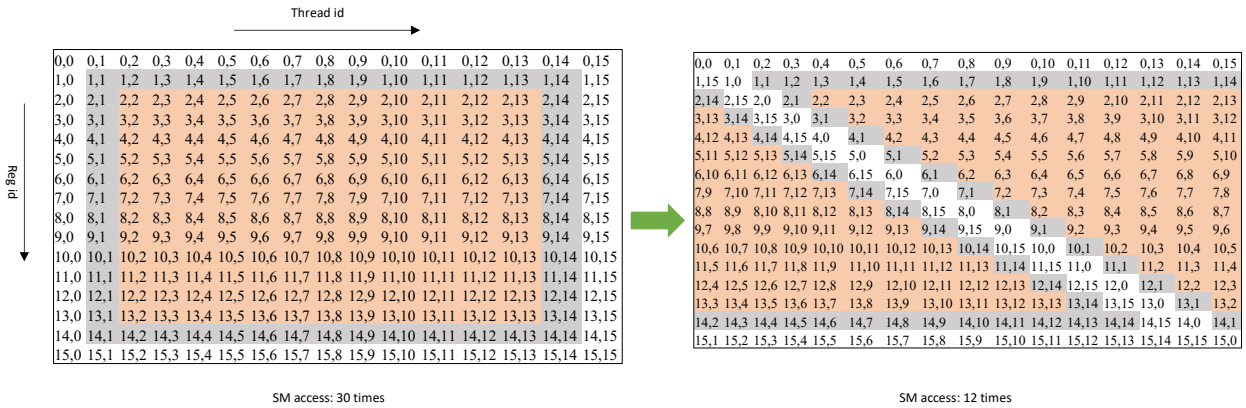


Figure 6.3: Shifting the cache region in Figure 6.2 for potential reduction in shared memory access.

6.3.3. Cache Management Across Different Hierarchy

The resource scaling optimizations we proposed in Section 5.4 are also not exclusive to GPUs. Our future work includes adapting the multi-circular queue data structure to accommodate arbitrary memory hierarchies. However, within the context of EBISU in this research, we operate under the assumption that the capacity of scratchpad memory is always sufficient, an assertion that isn't universally accurate. The capacity EBISU requires is largely influenced by the machine balance, as demonstrated in Section 5.9. A larger gap between memory bandwidth and compute implies a greater need for cache buffering space. Consequently, we aim to extend EBISU to leverage the space in scratchpad memory and L1 cache more efficiently. Plans to expand EBISU's functionality to use the L2 cache space are also underway. Figure 6.4 provides a rudimentary representation of how multi-queue can be ported to a memory hierarchy system.

Nevertheless, we foresee the potential for a more optimal solution that sustains the performance level of the L1 cache while augmenting the capacity to incorporate the L2 cache or even global memory.

6.4. Further Advancing Temporal Blocking

6.4.1. Multi-Level Temporal Blocking

EBISU currently only considers utilizing either with or without device tiling separately. However, applying device tiling on a temporal blocking kernel is another possibility. For example, imagine a kernel that employs temporal blocking with a depth of 2 and integrates device tiling with a depth of t.

Assuming an $\dot{t}$ for inner tiling, we establish $I' = \dot{t} \times I$ and $rad' = \dot{t} \times rad$. . Based on

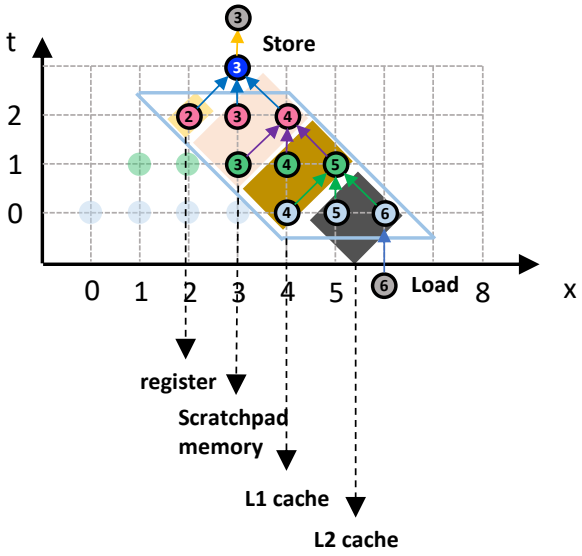


Figure 6.4: A preliminary concept for adapting multi-queue to exploit the hierarchy of memory systems

Equation 5.30, we get:

$$\begin{aligned} tile'_y = tile'_x > 4 \times rad' \times \mathbb{C}_{Cell} \times \frac{balance}{I'} &= 4 \times \dot{t} \times rad \times \mathbb{C}_{Cell} \times \frac{balance}{\dot{t} \times I} \\ &= 4 \times rad \times \mathbb{C}_{Cell} \times \frac{balance}{I} \end{aligned} \tag{6.1}$$

Furthermore, according to Equation 5.31, we have:

$$t \geq \frac{tile'_x \times 2}{\frac{I'}{balance \times \mathbb{C}_{Cell}} \times tile'_x - 4} = \frac{tile_x \times 2}{\frac{\dot{t} \times I}{balance \times \mathbb{C}_{Cell}} \times tile_x - 4} \tag{6.2}$$

From this, it is evident that even though two-level temporal blocking does not change the desired tiling, it can effectively reduce the desired depth for device tiling version temporal blocking. However, careful consideration is necessary to balance the overheads associated with the two types of temporal blocking.

6.4.2. Auto-Tuning or Performance Modeling with Varying Policies

EBISU is designed for a platform with sufficient cache capacity. However, in scenarios where cache capacity is constrained, we might need to oscillate between different policies. This could be accomplished through auto-tuning techniques or by developing a more sophisticated performance model incorporating cache capacity considerations.

Additionally, recent developments in machine learning [147] present the promising potential for parameter tuning, an aspect we are considering for future exploration.

6.5. Balancing Concurrency and Resources

This research directly uses the conclusion of Volkov's research [148]. We operate under the assumption that an increase in concurrency does not further impact performance beyond a certain point, and we rely on Little's Law to identify this point.

However, we have not thoroughly explored the trade-off space where changes in concurrency can impact performance. For example, at a specific juncture, would it be more advantageous to devote on-chip resources to increasing parallelism or caching data at a specific juncture? Or, under certain circumstances, might it be beneficial to trade off concurrency - reducing it to a point where parallelism is insufficient for effective device utilization - in favor of on-chip resources for caching?

We anticipate that a more precise performance model might further benefit the performance of memory-bound kernels.

6.6. On-chip Memory Capacity and Machine Imbalance

This research primarily concentrates on the impact of on-chip resources and their role in mitigating machine imbalance. It provides a new opportunity for hardware designs that capitalize on larger on-chip resources to utilize imbalanced machines. A key extension of this research would be to address the following question: What is the optimal on-chip memory capacity that delivers the best cost-performance ratio for a particular machine balance?

6.7. Load Balancing

Throughout this dissertation, we operate under the assumption of a perfectly balanced workload, an ideal that may not always hold in practical environments. We posit this study as a pioneering exploration, validating the feasibility of PERKS and EBISU. However, both PERKS and EBISU demand a certain level of manual workload scheduling, bypassing the dynamic scheduling capabilities of the hardware. In this context, there are two possible future directions: either hardware vendors may recognize the potential of this research and provide hardware support, or we undertake the investigation into a software-centric workload scheduling system. We believe Osama's research [149] can effectively complement this work. We view the incorporation of workload balancing into both PERKS and EBISU as a valuable direction for future work.

CHAPTER 7

Conclusion

This dissertation addresses the challenge of increasing imbalance in processors, i.e. the gap between compute and memory bandwidth is increasing over time, and we expect that this trend will continue in the near future.

Taking the latest hardware trends into consideration, this dissertation proposed two strategies to overcome the machine imbalance: we propose to extend the lifetime of the kernel across the time steps and take advantage of the large volume of on-chip resources in reducing or eliminating traffic to the device memory, and we propose to determine and stick to the minimize parallelism to maximize the available on-chip resources.

In this dissertation, we apply these two strategies in two lines of research. In PERKS, we applied the measurements to leverage on-chip resources to reduce the memory traffics by caching the intermediate results. In EBISU, we applied the strategies to optimize temporal blocking that usually suffers from resource-bound. As a result, these simple strategies benefit the performance of a wide range of memory-bound kernels significantly.

While this work provides valuable insights into memory-bound kernel optimizations, there remain many unexplored opportunities and directions for future research, which we hope this work will inspire.

Bibliography

[1] T. Zhao, P. Basu, S. Williams, M. Hall, and H. Johansen, "Exploiting reuse and vectorization in blocked stencil computations on cpus and gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–44.

[2] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, and P. Sadayappan, "Effective resource management for enhancing performance of 2d and 3d stencils on gpus," in *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, 2016, pp. 92–102.

[3] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[4] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L.-N. Pouchet, and P. Sadayappan, "Domain-specific optimization and generation of high-performance gpu code for stencil computations," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1902–1920, 2018.

[5] K. Matsumura, H. R. Zohouri, M. Wahib, T. Endo, and S. Matsuoka, "AN5D: automated stencil framework for high-degree temporal blocking on gpus," in *CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020*, 2020, pp. 199–211. [Online]. Available: https://doi.org/10.1145/3368826.3377904

[6] N. Maruyama and T. Aoki, "Optimizing Stencil Computations for NVIDIA Kepler GPUs," in *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*, A. Größlinger and H. Köstler, Eds., Vienna, Austria, Jan. 2014, pp. 89–95.

[7] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–23, 2013.

[8] P. Chen, M. Wahib, S. Takizawa, R. Takano, and S. Matsuoka, "A versatile software systolic execution model for gpu memory-bound kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–81.

[9] ——, "Efficient algorithms for the summed area tables primitive on gpus," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 482–493.

[10] ——, "Ifdk: A scalable framework for instant high-resolution image reconstruction," in *Proceedings of the International Conference for High Performance Computing, Networking,*

Storage and Analysis, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356163

[11] P. Chen, M. Wahib, X. Wang, T. Hirofuchi, H. Ogawa, A. Biguri, R. Boardman, T. Blumensath, and S. Matsuoka, "Scalable fbp decomposition for cone-beam ct reconstruction," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476139

[12] Nvidia, "Nvidia cuda sample," 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-samples/index.html

[13] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 678–689.

[14] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, and E. S. Quintana-Ortí, "Ginkgo: A modern linear operator algebra framework for high performance computing," 2020.

[15] Fujistu, "Fujitsu processor a64fx," 2023. [Online]. Available: https://www.fujitsu.com/downloads/SUPER/a64fx/a64fx_datasheet.pdf

[16] Intel, "App metrics for intel microprocessors," https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Xeon-Processors.pdf, 2023.

[17] cpu monkey, "Intel xeon platinum 8368 benchmark, test and specs," https://www.cpu-monkey.com/en/cpu-intel_xeon_platinum_8368/, 2023.

[18] H. systems, "Amd epyc™ 7003 series," https://www.hpc.co.jp/product/cpu_amd_epyc-7003-processors/, 2023.

[19] AMD, "Amd epyc™ 7773x," https://www.amd.com/en/products/cpu/amd-epyc-7773x/, 2023.

[20] Nvidia. (2021) Nvidia a100 tensor core gpu architecture. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf

[21] NVIDIA, "Nvidia h100 tensor core gpu architecture," 2022. [Online]. Available: https://www.hpctech.co.jp/catalog/gtc22-whitepaper-hopper_v1.01.pdf

[22] AMD, "Amd instinct™ mi200 series accelerator," https://www.amd.com/system/files/documents/amd-instinct-mi200-datasheet.pdf/, 2023.

[23] J. D. McCalpin *et al.*, "Memory bandwidth and machine balance in current high performance computers," *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, vol. 2, no. 19-25, 1995.

[24] K. Rupp, "Cpu, gpu and mic hardware characteristics over time," https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/, 2013.

[25] J. D. McCalpin, "The evolution of single-core bandwidth in multicore processors," http://sites.utexas.edu/jdm4372, 2023.

[26] J. Domke, E. Vatai, A. Drozd, P. ChenT, Y. Oyama, L. Zhang, S. Salaria, D. Mukunoki, A. Podobas, M. WahibT, and S. Matsuoka, "Matrix engines for high performance computing: A paragon of performance or grasping at straws?" in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 1056–1065.

[27] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel, "Applying the roofline model," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 76–85.

[28] K.-H. Kim, K. Kim, and Q.-H. Park, "Performance analysis and optimization of three-dimensional fdtd on gpu using roofline model," *Computer Physics Communications*, vol. 182, no. 6, pp. 1201–1207, 2011.

[29] J. Meng and K. Skadron, "A performance study for iterative stencil loops on gpus with ghost zone optimizations," *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 115–142, 2011.

[30] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, A. Rountev, L.-N. Pouchet, and P. Sadayappan, "On optimizing complex stencils on gpus," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 641–652.

[31] H. Courtecuisse and J. Allard, "Parallel dense gauss-seidel algorithm on many-core processors," in *2009 11th IEEE International Conference on High Performance Computing and Communications*. IEEE, 2009, pp. 139–147.

[32] M. Fratarcangeli, V. Tibaldo, and F. Pellacini, "Vivace: A practical gauss-seidel method for stable soft body dynamics," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 6, pp. 1–9, 2016.

[33] A. Kochurov and D. Golovashkin, "Gpu implementation of jacobi method and gauss-seidel method for data arrays that exceed gpu-dedicated memory size," *Journal of Mathematical Modelling and Algorithms in Operations Research*, vol. 14, no. 4, pp. 393–405, 2015.

[34] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High performance stencil code generation with lift," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 100–112.

[35] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim, "Hbm (high bandwidth memory) dram technology and architecture," in *2017 IEEE International Memory Workshop (IMW)*, 2017, pp. 1–4.

[36] NVIDIA, "V100 gpu architecture," 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[37] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature nanotechnology*, vol. 15, no. 7, pp. 529–544, 2020.

[38] 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*, ser. HPDC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 85–98. [Online]. Available: https://doi.org/10.1145/2600212.2600213

[39] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 27–39. [Online]. Available: https://doi.org/10.1109/ISCA.2016.13

[40] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: Insights from a micro-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.

[41] Y. Oh, G. Koo, M. Annavaram, and W. W. Ro, "Linebacker: Preserving victim cache lines in idle register files of gpus," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 183–196.

[42] S. Darabi, M. Sadrosadati, N. Akbarzadeh, J. Lindegger, M. Hosseini, J. Park, J. Gómez-Luna, O. Mutlu, and H. Sarbazi-Azad, "Morpheus: Extending the last level cache capacity in

gpu systems using idle gpu core resources," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 228–244.

[43] R. Venkatesan, S. G. Ramasubramanian, S. Venkataramani, K. Roy, and A. Raghunathan, "Stag: Spintronic-tape architecture for gpgpu cache hierarchies," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 253–264, 2014.

[44] J. Domke, E. Vatai, B. Gerofi, Y. Kodama, M. Wahib, A. Podobas, S. Mittal, M. Pericàs, L. Zhang, P. Chen, A. Drozd, and S. Matsuoka, "At the locus of performance: A case study in enhancing cpus with copious 3d-stacked cache," 2022.

[45] M. Evers, L. Barnes, and M. Clark, "The AMD Next Generation Zen 3 Core," *IEEE Micro*, vol. 42, no. 3, pp. 7–12, 2022.

[46] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization," in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, vol. 1. IEEE, 2009, pp. 579–586.

[47] Y. Tang, R. You, H. Kan, J. J. Tithi, P. Ganapathi, and R. A. Chowdhury, "Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2015, pp. 205–214.

[48] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes, "Multicore-optimized wavefront diamond blocking for optimizing stencil updates," *SIAM Journal on Scientific Computing*, vol. 37, no. 4, pp. C439–C464, 2015.

[49] K. Akbudak, H. Ltaief, V. Etienne, R. Abdelkhalak, T. Tonellot, and D. Keyes, "Asynchronous computations for solving the acoustic wave propagation equation," *The International Journal of High Performance Computing Applications*, vol. 34, no. 4, pp. 377–393, 2020.

[50] "Top500," 2021, [Online; accessed 27-Mar-2021]. [Online]. Available: https://www.top500.org/lists/top500/2020/11/

[51] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 311–320. [Online]. Available: https://doi.org/10.1145/2304576.2304619

[52] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus," in *Proceedings of the 23rd International Conference on*

Supercomputing, ser. ICS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 256–265. [Online]. Available: https://doi.org/10.1145/1542275.1542313

[53] P. Rawat, M. Kong, T. Henretty, J. Holewinski, K. Stock, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Sdslc: A multi-target domain-specific compiler for stencil computations," in *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, ser. WOLFHPC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2830018.2830025

[54] U. Bondhugula, V. Bandishti, and I. Pananilath, "Diamond tiling: Tiling techniques to maximize parallelism for stencil computations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 5, pp. 1285–1298, May 2017. [Online]. Available: https://doi.org/10.1109/TPDS.2016.2615094

[55] T. Grosser, A. Cohen, P. H. J. 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*, ser. GPGPU-6. New York, NY, USA: Association for Computing Machinery, 2013, p. 24–31. [Online]. Available: https://doi.org/10.1145/2458523.2458526

[56] T. Muranushi and J. Makino, "Optimal temporal blocking for stencil computation," *Procedia Computer Science*, vol. 51, pp. 1303–1312, 2015, international Conference On Computational Science, ICCS 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050915011230

[57] D. Callahan, J. Cocke, and K. Kennedy, "Estimating interlock and improving balance for pipelined architectures," *Journal of Parallel and Distributed Computing*, vol. 5, no. 4, pp. 334–358, 1988. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0743731588900020

[58] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[59] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2013.

[60] C. Yang, T. Kurth, and S. Williams, "Hierarchical roofline analysis for gpus: Accelerating performance optimization for the nersc-9 perlmutter system," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 20, p. e5547, 2020.

[61] A. Cobham, "Priority assignment in waiting line problems," *Journal of the Operations Research Society of America*, vol. 2, no. 1, pp. 70–76, 1954.

[62] J. L. Gustafson, *Little's Law*. Boston, MA: Springer US, 2011, pp. 1038–1041. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_79

[63] P. Hijma, S. Heldens, A. Sclocco, B. van Werkhoven, and H. E. Bal, "Optimization techniques for gpu programming," *ACM Comput. Surv.*, vol. 55, no. 11, mar 2023. [Online]. Available: https://doi.org/10.1145/3570638

[64] N. CUDA, "Cuda c programming guide," 2021, [Online; accessed 27-Mar-2021]. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[65] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU technology conference, GTC*, vol. 10. San Jose, CA, 2010, p. 16.

[66] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar, "Fine-grained synchronizations and dataflow programming on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 109–118.

[67] S. Xiao and W.-c. Feng, "Inter-block gpu communication via fast barrier synchronization," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–12.

[68] T. Sorensen, A. F. Donaldson, M. Batty, G. Gopalakrishnan, and Z. Rakamarić, "Portable inter-workgroup barrier synchronisation for gpus," in *ACM SIGPLAN Notices*, vol. 51, no. 10. ACM, 2016, pp. 39–58.

[69] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a next-generation open source framework for deep learning," in *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, vol. 5, 2015, pp. 1–6.

[70] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: ACM, 1988, pp. 319–329. [Online]. Available: http://doi.acm.org/10.1145/73560.73588

[71] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '89. New York, NY, USA: ACM, 1989, pp. 655–664. [Online]. Available: http://doi.acm.org/10.1145/76263.76337

[72] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 311–320. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304619

[73] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 235–244. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250761

[74] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 256–265. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542313

[75] M. E. Belviranli, P. Deng, L. N. Bhuyan, R. Gupta, and Q. Zhu, "Peerwave: Exploiting wavefront parallelism on gpus with peer-sm synchronization," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 25–35.

[76] H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, and M. Köhler, "Preconditioned krylov solvers on gpus," *Parallel Computing*, vol. 68, pp. 32–44, 2017.

[77] A. Frommer, K. Lund, and D. B. Szyld, "Block krylov subspace methods for functions of matrices," 2017.

[78] J. W. Pearson and J. Pestana, "Preconditioners for krylov subspace methods: An overview," *GAMM-Mitteilungen*, vol. 43, no. 4, p. e202000015, 2020.

[79] Y. Saad, "The origin and development of krylov subspace methods," *Computing in Science & Engineering*, vol. 24, no. 4, pp. 28–39, 2022.

[80] Nvidia, "Inside kepler," 2022. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

[81] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 235–246.

[82] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[83] J. Taylor, *Introduction to error analysis, the study of uncertainties in physical measurements*. University Science Books, 1997.

[84] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 2008, pp. 1–11.

[85] Nvidia, "Nvidia cuda runtime api," 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[86] M. Harris *et al.*, "Optimizing parallel reduction in cuda," *Nvidia developer technology*, vol. 2, no. 4, p. 70, 2007.

[87] J. Luitjens, "Faster parallel reductions on kepler," *Parallel Forall. NVIDIA Corporation. Available at: https://devblogs. nvidia. com/parallelforall/faster-parallel-reductions-kepler*, 2014.

[88] P. J. Martín, L. F. Ayuso, R. Torres, and A. Gavilanes, "Algorithmic strategies for optimizing the parallel reduction primitive in cuda," in *2012 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2012, pp. 511–519.

[89] W. A. R. Jradi, H. Nascimento, and W. S. Martins, "A fast and generic gpu-based parallel reduction implementation," in *2018 Symposium on High Performance Computing Systems (WSCAD)*. IEEE, 2018, pp. 16–22.

[90] S. G. De Gonzalo, S. Huang, J. Gómez-Luna, S. Hammond, O. Mutlu, and W.-m. Hwu, "Automatic generation of warp-level primitives and atomic instructions for fast and portable parallel reduction on gpus," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Press, 2019, pp. 73–84.

[91] Nvidia, "Cub library," 2019. [Online]. Available: https://nvlabs.github.io/cub

[92] A.-K. C. Ahamed and F. Magoulès, "Efficient implementation of jacobi iterative method for large sparse linear systems on graphic processing units," *The Journal of Supercomputing*, vol. 73, no. 8, pp. 3411–3432, 2017.

[93] E. Phillips and M. Fatica, "A cuda implementation of the high performance conjugate gradient benchmark," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 2014, pp. 68–84.

[94] J. I. Aliaga, J. Pérez, and E. S. Quintana-Ortí, "Systematic fusion of cuda kernels for iterative sparse linear system solvers," in *European Conference on Parallel Processing*. Springer, 2015, pp. 675–686.

[95] R. Couturier and S. Domas, "Sparse systems solving on gpus with gmres," *The journal of Supercomputing*, vol. 59, no. 3, pp. 1504–1516, 2012.

[96] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the nvidia turing t4 gpu via microbenchmarking," *arXiv preprint arXiv:1903.07486*, 2019.

[97] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," *CoRR*, vol. abs/1804.06826, 2018. [Online]. Available: http://arxiv.org/abs/1804.06826

[98] T. Endo, Y. Takasaki, and S. Matsuoka, "Realizing extremely large-scale stencil applications on gpu supercomputers," in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, 2015, pp. 625–632.

[99] C. Pearson, M. Hidayetoğlu, M. Almasri, O. Anjum, I.-H. Chung, J. Xiong, and W.-M. W. Hwu, "Node-aware stencil communication for heterogeneous supercomputers," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 796–805.

[100] Y. Idomura, T. Ina, Y. Ali, and T. Imamura, "Acceleration of fusion plasma turbulence simulations using the mixed-precision communication-avoiding krylov method," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, 2020, p. 93. [Online]. Available: https://doi.org/10.1109/SC41405.2020.00097

[101] M. Wahib and N. Maruyama, "Scalable kernel fusion for memory-bound GPU applications," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, 2014, pp. 191–202. [Online]. Available: https://doi.org/10.1109/SC.2014.21

[102] T. Gysi, T. Grosser, and T. Hoefler, "MODESTO: data-centric analytic optimization of complex stencil programs on heterogeneous architectures," in *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*, 2015, pp. 177–186. [Online]. Available: https://doi.org/10.1145/2751205.2751223

[103] Y. Fu, E. Bolotin, N. Chatterjee, D. Nellans, and S. W. Keckler, "Gpu domain specialization via composable on-package architecture," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 1, dec 2021. [Online]. Available: https://doi.org/10.1145/3484505

[104] J. D. C. Little, "A proof for the queuing formula: L = λw," *Operations Research*, vol. 9, pp. 383–387, 1961.

[105] X. Mei, K. Zhao, C. Liu, and X. Chu, "Benchmarking the memory hierarchy of modern gpus," in *IFIP International Conference on Network and Parallel Computing*. Springer, 2014, pp. 144–156.

[106] L. Zhang, M. Wahib, H. Zhang, and S. Matsuoka, "A study of single and multi-device synchronization methods in nvidia gpus," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 483–493.

[107] B. van Werkhoven, "Kernel tuner: A search-optimizing gpu code auto-tuner," *Future Generation Computer Systems*, vol. 90, pp. 347–358, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18313359

[108] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[109] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[110] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, "Zorua: A holistic approach to resource virtualization in gpus," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–14.

[111] C. Li, Y. Yang, Z. Lin, and H. Zhou, "Automatic data placement into gpu on-chip memory resources," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 23–33.

[112] P. Micikevicius, "3d finite difference computation on gpus using cuda," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: Association for Computing Machinery, 2009, p. 79–84. [Online]. Available: https://doi.org/10.1145/1513895.1513905

[113] N. CUDA, "Cuda toolkit documentation," *NVIDIA Developer Zone. http://docs.nvidia.com/cuda/index.html*, 2020.

[114] P. R. David Eberius, David Rogers, "Understanding strong scaling on gpus using empirical performance saturation size," in *The International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. International Workshop on Performance Portability and Productivity (P3HPC), 2022.

[115] H. Anzt, Y. M. Tsai, A. Abdelfattah, T. Cojean, and J. Dongarra, "Evaluating the performance of nvidia's a100 ampere gpu for sparse and batched computations," in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2020, pp. 26–38.

[116] N. Beams, A. Abdelfattah, S. Tomov, J. Dongarra, T. Kolev, and Y. Dudouit, "High-order finite element method using standard and device-level batch gemm on gpus," in *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2020, pp. 53–60. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ScalA51936.2020.00012

[117] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *Proc. of the IEEE IPDPS'10*. Atlanta, GA: IEEE Computer Society, April 19-23 2010, pp. 1–8, DOI: 10.1109/IPDPSW.2010.5470941.

[118] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on gpus," in *Proceedings of the conference on high performance graphics 2009*, 2009, pp. 145–149.

[119] K. Gupta, J. A. Stuart, and J. D. Owens, *A study of persistent threads style GPU programming for GPGPU workloads*. IEEE, 2012.

[120] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao, "Dynamic load balancing on single- and multi-gpu systems," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–12.

[121] F. Daoud, A. Watad, and M. Silberstein, "Gpurdma: Gpu-side library for high performance networking from gpu kernels," in *Proceedings of the 6th international Workshop on Runtime and Operating Systems for Supercomputers*, 2016, pp. 1–8.

[122] C. Jung, S. Kim, I. Yeom, H. Woo, and Y. Kim, "Gpu-ether: Gpu-native packet i/o for gpu applications on commodity ethernet," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[123] F. Khorasani, H. Asghari Esfeden, N. Abu-Ghazaleh, and V. Sarkar, "In-register parameter caching for dynamic neural nets with virtual persistent processor specialization," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 377–389.

[124] F. Zhu, J. Pool, M. Andersch, J. Appleyard, and F. Xie, "Sparse persistent rnns: Squeezing large recurrent networks on-chip," *arXiv preprint arXiv:1804.10223*, 2018.

[125] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of computational Physics*, vol. 53, no. 3, pp. 484–512, 1984.

[126] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM review*, vol. 51, no. 1, pp. 129–159, 2009.

[127] M. Wahib and N. Maruyama, "Automated gpu kernel transformations in large-scale production stencil applications," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 259–270. [Online]. Available: https://doi.org/10.1145/2749246.2749255

[128] H. Stengel, J. Treibig, G. Hager, and G. Wellein, "Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 207–216.

[129] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[130] T. Endo, "Applying recursive temporal blocking for stencil computations to deeper memory hierarchy," in *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2018, pp. 19–24.

[131] H. R. Zohouri, A. Podobas, and S. Matsuoka, "Combined spatial and temporal blocking for high-performance stencil computation on fpgas using opencl," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 153–162. [Online]. Available: https://doi.org/10.1145/3174243.3174248

[132] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege, "Hybrid hexagonal/classical tiling for gpus," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014, pp. 66–75.

[133] T. Grosser, S. Verdoolaege, A. Cohen, and P. Sadayappan, "The relation between diamond tiling and hexagonal tiling," *Parallel Processing Letters*, vol. 24, no. 03, p. 1441002, 2014. [Online]. Available: https://doi.org/10.1142/S0129626414410023

[134] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama, "Opencl-based fpga-platform for stencil computation and its optimization methodology," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1390–1402, 2016.

[135] A. Podobas, K. Sano, and S. Matsuoka, "A template-based framework for exploring coarse-grained reconfigurable architectures," in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2020, pp. 1–8.

[136] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 2008, pp. 1–12.

[137] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, A. Rountev, L.-N. Pouchet, and P. Sadayappan, "On optimizing complex stencils on gpus," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 641–652.

[138] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-d blocking optimization for stencil computations on modern cpus and gpus," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–13.

[139] P. S. Rawat, F. Rastello, A. Sukumaran-Rajam, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Register optimizations for stencils on gpus," *SIGPLAN Not.*, vol. 53, no. 1, pp. 168–182, Feb. 2018. [Online]. Available: http://doi.acm.org/10.1145/3200691.3178500

[140] X. You, H. Yang, Z. Jiang, Z. Luan, and D. Qian, "Drstencil: Exploiting data reuse within low-order stencil on gpu," in *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. Los Alamitos, CA, USA: IEEE Computer Society, dec 2021, pp. 63–70. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HPCC-DSS-SmartCity-DependSys53884.2021.00036

[141] L. Zhang, M. Wahib, P. Chen, J. Meng, X. Wang, T. Endo, and S. Matsuoka, "Perks: A locality-optimized execution model for iterative memory-bound gpu applications," in *Proceedings of the 37th International Conference on Supercomputing*, ser. ICS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 167–179. [Online]. Available: https://doi.org/10.1145/3577193.3593705

[142] L. Yuan, H. Cao, Y. Zhang, K. Li, P. Lu, and Y. Yue, "Temporal vectorization for stencils," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476149

[143] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector simd architectures," in *International Conference on Compiler Construction*. Springer, 2011, pp. 225–245.

[144] X. Liu, Y. Liu, H. Yang, J. Liao, M. Li, Z. Luan, and D. Qian, "Toward accelerated stencil computation by adapting tensor core unit on gpu," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–12.

[145] T. Zhao, S. Williams, M. Hall, and H. Johansen, "Delivering performance-portable stencil computations on cpus and gpus using bricks," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2018, pp. 59–70.

[146] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[147] L. Trümper, T. Ben-Nun, P. Schaad, A. Calotoiu, and T. Hoefler, "Performance embeddings: A similarity-based transfer tuning approach to performance optimization," in *Proceedings of the 37th International Conference on Supercomputing*, ser. ICS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 50–62. [Online]. Available: https://doi.org/10.1145/3577193.3593714

[148] V. Volkov, *Understanding latency hiding on GPUs*. University of California, Berkeley, 2016.

[149] M. Osama, "Gpu load balancing," 2022.