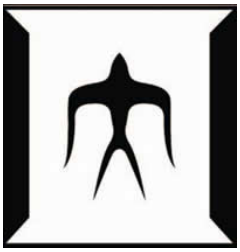
論文 / 著書情報
Article / Book Information

題目(和文)	GPU上の仮想シストリックアレイ
Title(English)	A Virtual Systolic Array on GPUs
著者(和文)	陳 鵬
Author(English)	Peng Chen
出典(和文)	学位:博士(学術), 学位授与機関:東京工業大学, 報告番号:甲第11530号, 授与年月日:2020年3月26日, 学位の種別:課程博士, 審査員:松岡 聡,遠藤 敏夫,額田 彰,脇田 建,横田 理央
Citation(English)	Degree:Doctor (Academic), Conferring organization: Tokyo Institute of Technology, Report number:甲第11530号, Conferred date:2020/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

A Virtual Systolic Array on GPUs



Peng Chen

Supervisor: Prof. Satoshi Matsuoka

Department of Mathematical and Computing Science
Tokyo Institute of Technology

This dissertation is submitted for the degree of
Doctor of Philosophy

School of Computing March 2020

I would be greatly thankful to my wife and my family for their continuous support in my life. I would like to thank my supervisor Professor Satoshi Matsuoka who offered me the great opportunity to enroll in the AIST-Tokyo Tech program so that I had the chance to join the Matsuoka-lab at Tokyo Institute of Technology. I also greatly thank all of the members in the RWBC-OIL (AIST-Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory): Director Hirotaka Ogawa, Group Leader Ryousei Takano, Senior Researcher Mohamed Wahib, Visiting Researcher Shinichiro Takizawa, Postdoc Nguyen Truong, Postdoc Hiroki Kanezashi, Postdoc Poiyapram Vinayaraj. I would like to especially thank the Senior researcher Mohamed Wahib. His sharp insights and comments always motivate me to find better solutions. With his great help, I improve my mind and skills little by little in my research road. I would like to especially thank Dr. Shinichiro Takizawa and Dr. Ryousei Takano who give a lot of precious advice on my research, e.g. presentation, paper writing. Thanks a lot to all of the members in Matsuoka-lab: Keiko Yoshida, Akihiro Nomura, Aleksandr Drozd, Jens Domke, Yosuke Oyama, Yohei Tsuji, Lingqi Zhang, Keita Yashima, Toshiki Tsuchikawa. I would like to thank Endo Lab at Tokyo Institute of Technology for providing a Tesla V100 GPU Server. I am grateful to Prof. Toshio Endo, Associate Prof. Akira Nukada, and Associate Prof. Rio Yokota for their guidance and advice in this research.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 28,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 100 figures.

Peng Chen
March 2020

Publications

Refereed

1. Peng Chen, Mohamed Wahib, Shinichiro Takizawa, Ryousei Takano, and Satoshi Matsuoka. 2018. **Efficient Algorithms for the Summed Area Tables Primitive on GPUs**. In 2018 IEEE International Conference on Cluster Computing (CLUSTER).482–493.

2. Peng Chen, Mohamed Wahib, Shinichiro Takizawa, Ryousei Takano, and Satoshi Matsuoka. 2019. **iFDK: A Scalable Framework for Instant Highresolution Image Reconstruction**. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19). ACM, New York, NY, USA, Article 84, 24 pages.

3. Peng Chen, Mohamed Wahib, Shinichiro Takizawa, Ryousei Takano, and Satoshi Matsuoka. 2019. **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 (SC '19). ACM, New York, NY, USA, Article 53, 81 pages. (Nominated as BP and BSP finalist)

Non-Refereed

1. Peng Chen, Mohamed Wahib, Shinichiro Takizawa, and Satoshi Matsuoka. **Pushing the Limits for 2D Convolution Computation on CUDA-enabled GPUs**. Technical Report 22, HPC163.

2. Peng Chen, Mohamed Wahib, Shinichiro Takizawa, Ryousei Takano, and Satoshi Matsuoka. **A Software Systolic Array on GPUs**. GPU Technology Conference Silicon Vally 2020 (GTC' 20), Poster.

3. Peng Chen, Mohamed Wahib, Shinichiro Takizawa, Ryousei Takano, Hirotaka Ogawa, and Satoshi Matsuoka. **High-resolution Image Reconstruction on Supercomputer**. GPU Technology Conference Silicon Vally 2020 (GTC' 20), Poster.

Abstract

Driven by the critical demands of intensive computation in a variety of fields, an old and heavily-researched parallel computer architecture, namely systolic array, is being revived in the post-Moore's law era, e.g. Google's Tensor Processing Unit (TPU), Nvidia's Tensor Core. Such a kind of domain-specific processors can provide extremely high TOPS (TeraOps/Second) and TOPS/Watt for compute-intensive algorithms, i.e. convolution, dense matrix multiplication, AI-purposed computation, Discrete Cosine Transform (DCT). In the last decade, we witnessed the emergence of Graphics Processing Units (GPU). As one of the most popular accelerators recently, GPUs are adopted to speed up the computation in a wide range of fields, i.e. scientific, engineering. Taking the latest TOP500 rankings in High-Performance Computing (HPC) for instance, more than half of the FLOPS comes from the Nvidia Tesla GPUs.

A systolic array is a network of processing elements (PEs) that rhythmically compute, accumulate and transfer data through the system. In typical systolic arrays, all of the PEs are often nested as a two-dimensional mesh. The systolic array architecture is simple yet very effective to achieve both computation and energy efficiencies with very limited memory bandwidth. Inspired by the mechanism of the hard-wired systolic arrays, we innovate a versatile execution model on the top of CUDA architecture for optimizing applications on GPUs. More specifically, we mimic the behavior of systolic arrays by CUDA warp, register files, and shuffle instruction. It is noteworthy that the three key techniques contribute to our model. First, register cache; second, partial sums accumulation and transfer; third, in-register computation. Our model improves the performance of regular memory-bound kernels by taking advantage of thread-private registers as a cache to perform the efficient in-register computation and employing the shuffle intrinsic to exchange partial sums between CUDA threads within a single CUDA Warp. Our model can be viewed as Software Systolic Array Execution Model (SSAM). In other words, we build a virtual systolic array on the top of CUDA architecture.

Regarding most of the scientific applications (or kernels), the increasing flops-to-bytes ratio of GPUs makes their computational performances bounded by the memory bandwidth. Memory-bound kernels that have a regular pattern of computation are particularly challenging

since they appear to be simple, yet they require very complex data reuse schemes to effectively utilize the CUDA memory hierarchy, e.g. global memory, shared memory, register memory, etc. Typically, advanced GPU implementations for memory-bound kernels on structured grids rely on the optimized use of fast on-chip scratchpad memory: the programmer uses this user-managed scratchpad memory for reducing the global memory access. Indeed, there exists a plethora of work proposing variations and combinations of the three locality schemes that rely on scratchpad memory: spatial blocking, temporal blocking, and a wavefront pipeline. Those complex locality schemes enabled strides in performance improvements. However, they essentially moved the bottleneck from the global memory to the faster, yet smaller, scratchpad. The objective of this study is to yet again move the bottleneck from the scratchpad to a faster resource: register files. Hence, we prioritize to use register files to cache data rather than shared memory, resulting in better data locality and higher computational efficiency.

A wide class of memory-bound kernels (or applications) can benefit from this execution model. In this study, we focus on mapping typical memory-bound kernels in SSAM, i.e. 2D convolution, 2D/3D stencils, Summed Area Tables (SAT). There is a strong motivation to optimize these kernels on GPUs. The 2D convolution and SAT computation become increasingly important in the Deep Learning workload. The stencil is a fundamental computation pattern in most of the scientific applications. To further improve the dependency graph in SSAM, we propose a novel algorithm to transpose the register cache locally (termed as BRLT). Unlike 2D convolution and stencils, we propose a collection of SSAM-based algorithms for SAT computation by the BRLT algorithm. To insight the performance advances of SSAM-based algorithms, we build performance models to analyze their mechanisms while using micro-benchmarking to measure some constant variables of GPUs.

The contributions in this study are as follows. **First**, we design and formulate a software systolic array on the top of CUDA architecture for speeding up the computation of memory-bound kernels with regular access on GPUs. **Second**, we analyze the data reuse and redundancy schemes to quantify the efficiency and limitations of SSAM. **Third**, we evaluate the proposed model for a wide variety of 2D/3D stencils, 2D general convolution, and SAT kernels on the latest Nvidia Tesla GPUs and demonstrate that SSAM-based algorithms outperform the top reported state-of-the-art libraries, e.g. NPP, Arrayfire, OpenCV. **Fourth**, we propose a novel algorithm to transpose the register matrix (used as register cache) locally and concurrently, resulting in improving the program dependency graph in SSAM for better locality and higher parallelism. **Finally**, we pave a novel way for code automation on GPUs due to the versatility and simplicity of SSAM for optimizing a large variant of algorithms.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

In the past decades, the development of the semiconductor industry followed the Moore's Law, which was formulated to predict the trend of manufacture in two aspects : on one hand, the density of the transistors approximately become double in every two years; on the other hand, the frequency (or the speed of the clock) may be doubled in every 18-24 months. However, in recent years, Moore's Law is significantly slowing down or ending for many semiconductor segments. In other words, we are already in the era of Post Moore's Law due to we reach the physical limits of both processors clock and power consumption as in literature [16, 153].

In recent years, to scale the performance of the processors, several processors employed manycore architecture and were emerged to provide increasingly computing throughputs, i.e. Intel Xeon Phi [144], Nvidia GPUs. Note that GPUs were originally designed for rendering real-time images and became a popular programmable device recently, known as General-Purpose computing on Graphics Processing Units (GPGPU). Driven by the critical demanding of Big Data workload, GPUs become one of the most popular accelerators to speed up such kind of computing-intensive workload. Taking the latest TOP500 rankings in High-Performance Computing (HPC) for instance, over half of the additional FLOPS is contributed by the Nvidia Tesla GPUs.

To compete with GPUs for optimizing the Deep Learning applications, systolic array, an old parallel processing architecture, is adopted for providing extremely high TOPS (TeraOps/Second) and TOPS/Watt [66], e.g. Google Tensor Processing Unit (TPU) [66]. and Nvidia Tensor Cores. The typical systolic array is a well-organized network with multiple processing elements. The architecture of systolic array is regular and simple, yet it can achieve very high computing performance, i.e. Google claimed that the TPU is an order of magnitude faster than Nvidia GPUs. Despite the outstanding performance is available to be achieved by the systolic array as mentioned earlier, the disadvantage of the systolic array can

not be ignored. More specifically, the systolic arrays are domain-specified hardware, taking TPU and Tensor Core for example, they can only perform the matrix-multiplication for Deep Learning workload. Furthermore, Nvidia Tensor core can only perform computations using half-precision.

Inspired by the hardwired systolic array, we innovate to build a virtual systolic array on the top of CUDA architecture, targeting to optimize a class of memory-bound kernels. Hence, we formulate a software systolic execution model (termed as SSAM) and illustrate the mechanism of our model by mapping several typical memory-bound algorithms. It is worth mentioning that we focus on memory-bound kernels with regular memory access patterns in this study. Kurzak et al. [83] developed a virtual systolic array for the QR decomposition while mapping their algorithm to a large distributed memory system. Such a virtual systolic architecture offers a simple, yet effective, computational model. However, it is specified for the strong scaling of the QR decomposition at thousand core-count regimes.

The remaining part of this chapter is organized as follows. In section 1.1, we introduce the background. Section 1.2 describes the challenges. Section 1.3 shows an overview of the proposed algorithms. In Section 1.4, we present the achieved performance. In Section 1.5, we list our contributions. Finally, Section 1.6 concludes.

1.1 Background

In this section, we firstly introduce the background of Nvidia CUDA. Then, we illustrate the motivation of employing register cache rather than shared memory cache. Next, we present the shuffle intrinsic that was used for efficient intra-warp communication. Finally, we show the bases of systolic array architecture.

1.1.1 Nvidia CUDA

In recent years, GPUs become one of the most important hardware components in computer systems. Initially, the purpose of a GPU (or video card) was to take a stream of binary data from CPU and create realtime images for output to display. However, modern GPUs are engaged as programmable devices for the most complex computations, e.g. big-data, data mining, and AI. Over several decades, the GPUs evolved from a single-core and function-specified hardware (solely used for graphics) to a collection of programmable many-cores. Nvidia Compute Unified Device Architecture (CUDA) is a parallel computing platform and application programming model.

Execution hierachy In this section, we introduce the Execution hierarchy of Nvidia GPUs. Nvidia GPU is built on a collection of multi-threaded Streaming Processors (SMs) running thousands of threads. Massive thread-level parallelism is running in a single instruction multi-thread (SIMT) model. The threads in CUDA are grouped into warps, blocks, and grids. Thousands of threads are created, scheduled, and executed concurrently. Unlike the latency-optimized CPUs, GPUs are a throughput-optimized processor. More specifically, GPUs are capable of executing a massive number of threads concurrently.

CUDA memory hierarchy This section presents the CUDA memory hierarchy. To approach the peak performance of GPU, it is essential to implement applications that efficiently utilize CUDA's memory hierarchy, i.e. global, shared, constant, and registers memory. Each thread has a set of dedicated register files, which is the fastest on-chip memory. Threads within a CUDA block share a fast, block-private scratchpad (namely shared memory). The performance characteristics of shared memory are similar to L1-cache due to L1-cache shares the space with shared memory. Shared memory uses 32 memory banks and thus avoiding bank conflict is essential to the throughput of shared memory access. In the same kernel, all threads share global GPU memory. Accesses to the global memory are cached via L1 and L2 caches.

1.1.2 Motivation of using register cache

Table 1.1 Shared Memory and Register Files on GPUs

Tesla GPU	Shared Memory/SM	32-bit registers/SM	SMs
K40	16/32/48 KB	65536	15
M40	96 KB	65536	24
P100	64 KB	65536	56
V100	up to 96 KB	65536	80

This section shows the strong motivation of using register files as a cache to optimize applications by comparing the details of registers and scratchpad. Register cache is an approach in which a single warp builds a virtual cache layer on top of register files for low-latency data access [8]. As Table 1.1 shown, in the latest GPUs, the capacity of register memory per SM is 256KB (65536*4B) and is more than ($\frac{256KB}{96KB} \approx$)2.7 times larger than shared memory.

Using register files as a cache can greatly improve applications' performance. Especially, since the number of SMs in modern GPU generations has increased, the gap between the

capacity of shared memory and registers files is becoming larger. It is important to mention that careful design of algorithms is required to avoid bank conflicts when using shared memory. Otherwise, the performance may drop dramatically.

To avoid overemphasize the use of register memory, it is important to note that we also take advantage of scratchpad as a cache to increase the data reuse and employ scratchpad to perform the inter-warp communication within a CTA.

1.1.3 CUDA shuffle intrinsic

CUDA provides efficient intra-warp communication intrinsics that is called shuffle. Shuffle intrinsics bring another way of sharing data among threads within the same CUDA Warp. The purpose of shuffle intrinsics is to read the specified registers which are held by other threads within the same Warp. More specifically, without using the shared or global memory, threads in a single Warp can exchange registers with each other directly. In terms of computing efficiency, using shuffle for in-register computation is superior in performance to the other memory types, due to its low latency and high throughput [22]. However, using these intrinsics has its limits that is related to the characteristics of the CUDA architecture, e.g. at most 32 threads (namely a single Warp) can be performed by the shuffle intrinsic call, introducing register pressure since the number of used registers increases, CUDA kernels become more intricate due to the use of the thread and lane indexing.

1.1.4 Systolic Array architecture

Typically, a systolic array consists of a large homogeneous network of primitive processing elements (PEs) which can be hardwired or software configured for a specific computational task. In other words, the systolic array, a parallel computer architecture, is built as a monolithic network of tightly coupled data processing elements (or units). As introduced in literature [126, 20, 80, 81, 44], the systolic array architecture was invented by H. T. Kung and Charles Leiserson and successfully applied for many dense linear algebra computations (banded matrices), e.g. solving systems of linear equations, matrix multiplication, LU decomposition. In the past decades, systolic arrays have been well researched. The authors in [77, 129, 169, 92, 23] have proposed well-structured systolic arrays architectures. A hardware systolic array is a computing pipeline network with physical interconnects between PEs. As Figure 1.1 shows, all of the PEs are connected with their neighbors. The PEs compute, store, and transfer their partial results to downstream neighbors in parallel.

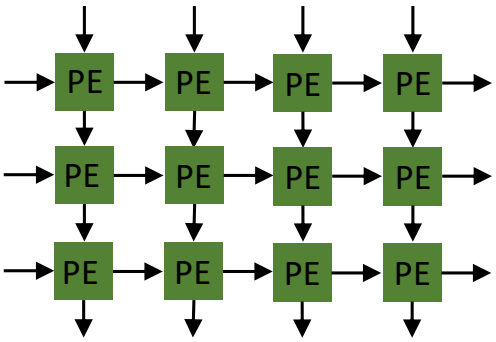


Fig. 1.1 Hardware systolic array.

1.2 Problem statement

GPU accelerators have been increasingly adopted to meet the exponentially growing computational requirements in various fields, such as scientific simulations and machine learning. The systolic array is an old yet revived parallel computing architecture. Recently, it becomes more and more attractive to be used for speeding up the compute-intensive applications, i.e. Deep Learning. We innovate a virtual systolic on GPUs (namely implement a systolic array using the software) to solve the commonly occurring computational motifs in HPC (and occasionally deep learning): memory-bound regular computation on a structured grid [2]. Memory-bound kernels that have a regular pattern of computation are particularly challenging, since they appear to be simple, yet they require very complex data reuse schemes to effectively utilize the memory hierarchy. Typically, advanced GPU implementations for memory-bound kernels on structured grids rely on the optimized use of fast on-chip scratchpad memory: the programmer uses this user-managed scratchpad memory for reducing the global memory access. Indeed, there exists a plethora of work proposing variations and combinations of the three locality schemes that rely on scratchpad memory: spatial blocking, temporal blocking, and a wavefront pipeline (the reader can find some of the notable work at [101, 106, 114, 146, 170, 100]). Those complex locality schemes enabled strides in performance improvements. However, they essentially moved the bottleneck from the global memory to the faster, yet smaller, scratchpad. The objective of this work is to yet again move the bottleneck from the scratchpad to a faster resource (namely register files) while building a general model to leverage our methodology. To our best knowledge, this is the first work to employ a general and versatile model to optimize applications on CUDA-enabled GPUs. The following are some of the challenges one has to consider when building a model that mimics the mechanism of a systolic array.

1.2.1 Challenges in CUDA

The following are some of the challenges one has to consider when optimizing algorithms on Nvidia GPUs.

(I) To achieve the peak performance of GPUs, it is essential to fully take advantage of the complex memory hierarchy, e.g. global memory, shared memory, register files.

(II) Global memory is the largest but slowest off-chip memory, it is critical to design suitable data layout (or placement) to meet the requirement of the caching mechanism of L1/L2.

(III) Multiple CUDA threads on GPUs are managed as Warps, each Warp (32 threads execute in a group) performs operations in a SIMD (Single Instruction Multiple Data) fashions. Suitable tiling (or blocking) method benefits to increase the computational parallelism and decrease the intra-warp divergence.

(IV) To tolerate the latency of streaming processors, sufficient concurrency is fundamentally required to meet the number of physical cores.

1.2.2 Challenges in building a virtual systolic array

Before building a virtual systolic array on GPUs, it is required to understand the mechanism of the hardware systolic array. The architecture of the systolic array is simple and regular, yet it is not easy to exchange register data between threads as the PEs in a systolic array. we rely on the Warp shuffle primitives that provide low-latency register exchange within a single Warp [119] as that different threads in a Warp compute the partial sums, before moving the partial sums to the downstream neighbor thread to be accumulated.

To match the high throughput of shuffling the partial sums, we fully utilize the registers for caching the computed partial sums. Accordingly, our model can perform structured grid memory-bound computations at low latency and high throughput. As a result, we can decrease the dependency on scratchpad or cache memory and thus improve the application's performance by avoiding the scratchpad and cache bottleneck. To avoid overemphasis on intra-warp communication, it is necessary to clarify that we do not limit the use of scratchpad for inter-warp communication.

Kernels that can be mapped in SSAM should have a regular memory access pattern. However, several challenges should be addressed to implement an efficient software systolic model on the top of partial sums accumulation/exchange, and register caching. First, the algorithms must be expressed in the way the systolic array can process. Second, the total

number of threads executed together in a working unit (i.e. CUDA warp) is relatively small[1]. Hence, this enforces a limit on the systolic array size and parallelism. We have to rely on Instruction Level Parallelism (ILP) and in-thread data reuse to provide enough concurrency in each node (i.e. thread) in the systolic array. Finally, the shuffle primitives work only for a single warp: a redundancy method for halo layers, along with a redundancy analysis, is necessary.

1.2.3 Challenges in building performance model

To understand the characteristics of the proposed systolic array model, a performance model is fundamentally required for the theoretical performance analysis. It is important to point out that most of the applications could be mapped to the virtual systolic array in the same pattern, yet the details of the analysis may vary from case to case. More specifically, we need to create micro-benchmarking to measure the constant parameters of GPUs, e.g. the latency of shuffle intrinsic and arithmetical pipeline. According to the proposed algorithm, we also need to formulate the performance model to investigate the mechanism of the proposed algorithms, i.e. understanding the impact of different parameters on the performance of our virtual systolic array. Thereby, we can justify the validation of achieved performance via such a performance model.

1.3 Proposed algorithms

In this study, we propose a versatile systolic execution model (namely SSAM) for improving the performance of memory-bound kernel with regular access patterns. The reader can find a detailed illustration of SSAM in Chapter 3. On CUDA-enabled GPUs, a wide class of kernel and applications can benefit from this model, i.e. scan/reduction operator, Summed Area Tables, convolution, stencils, etc. The systolic array model is based on the transfer and accumulation of partial results (or summations) in the thread-private registers. Meanwhile, we take advantage of the register files as a user-managed cache to avoid using the scratchpad altogether. As introduced by Ben-Sasson et al. [9], there are several advantages of employing such a warp-centric approach :

- Divergence-free warps. The cost of divergence between warp is very lightweight.
- SIMD-optimized warp execution. All threads in a single warp perform the same instruction in a lock-step.

[1] *WarpSize* is equal to 32 on all Nvidia GPU generations

- Non-synchronization between threads within a warp. No requirement of synchronization between threads in a warp. we can avoid __syncthreads overhead since it is often required when using shared memory.

To leverage such a warp-centric approach, the SSAM can simplify the designation of algorithms. Register cache can be a general technique that can replace shared memory with the shuffle. As concluded by Ben-Sasson et al. the use of register cache can improve application performance by 50% in his experiment.

1.3.1 Register cache

Register cache [85, 41, 59, 8] is a widely used approach to optimize algorithm on GPUs. To decrease the global memory access and increase the data reuse in fast on-chip memory, we employ register files to cache data that are moved from/to the global memory and thus, we can perform the efficient in-register computations and operations as will be addressed in later paragraphs.

To avoid overemphasize the use of register memory, it is important to note that we also employ the shared memory as a cache to increase the data reuse and perform the inter-warp communication within a CUDA block. Note that using shared memory as a user-managed cache is the vendor recommended programming model. Taking the 2D convolution for instance, the input image data is cached by registers, the filter weights are cached by shared memory; Using the collection of proposed SAT implementations as examples, the shared memory is used to accumulate the partial sums, it is also used to transpose the register matrix of size 32×32. In the stencil computation implementation, the shared memory is employed to cache the partial results and performs inter-warp communication.

1.3.2 Partial sums computation and transfer

All registers in SSAM can mimic the functions of PEs in the hardwired systolic array as mentioned earlier. Different threads in a warp compute the partial results (or sums), move the partial sums to the downstream neighbor thread as a PE in the systolic array for further accumulation. Importantly, the partial sums that are held by registers can be exchanged within a single warp efficiently.

To accumulate and transfer partial sums using thread-private registers, in a SIMT fashion, different groups of threads operate over different input points, with some data redundancy that we introduce to account for the halo layers. All threads in a single Warp performs in a SIMD fashion, the register cache strategy also benefits to the global memory access since their access pattern meet the requirement of L1 and L2 cache on GPUs.

Within a single thread, there is not any cost to transfer the partial sum due to the registers can naturally be accessed by the specified thread; Within a single Warp, the thread-private registers can be accessed by other threads by the lightweight cost of using shuffle intrinsic.

In order to transfer the partial sums efficiently and increase the computational parallelism, we innovate a novel algorithm (named as BRLT in chapter 6) to improve the dependency of the algorithm for SSAM by transposing the register matrix locally. That being said, we can further improve the performance of the SSAM-optimized applications by tuning their algorithmic dependencies.

1.3.3 Efficient in-register computation

Similar to the hardware systolic array, the proposed software systolic array is also capable of optimizing algorithms to achieve a very high TOP/s, as will be presented in the later sections. The advance of using register cache is that we can perform in-register computation efficiently as much as possible due to all of the data and partial sums are held by register memory, which is the fastest on-chip memory. In other words, we can fully use the advance of registers: High-throughput and Low-latency.

1.4 Results

In this study, we map three typical kernels in SSAM, namely 2D convolution, stencils, and SAT, which are typical memory-bound kernels with regular memory access on GPUs. Using the latest Tesla P100 and V100 GPUs in a single compute node, the SSAM-based algorithms are evaluated in detail as will be illustrated in the later chapters. We use the same SSAM model to optimize all algorithms, yet the details vary from algorithm to algorithm. It is because the dependencies of all algorithms are distinct. Regarding the SAT implementation, we proposed a collection of SSAM-based algorithms that adjust the dependencies with each other.

1.5 Contributions

In this study, we propose a novel and versatile systolic array execution model on GPUs. We also optimize a collection of typical memory-bound kernels by mapping those algorithms in SSAM. The contributions of this study are as follows:

- Inspired by the hardwired systolic arrays architecture, we illustrate A formulation, design, and implementation of a virtual systolic model on GPUs (called SSAM) for efficiently computing memory-bound kernels with regular access pattern.
- We present a detailed analysis of the data reuse and redundancy schemes to quantify the efficiency and limitations of SSAM.
- Evaluation of the proposed model for a wide variety of iterative 2D/3D stencils and 2D general convolution on Nvidia Tesla P100 and V100 GPUs. Our model outperforms the top reported state-of-the-art implementations, including implementations with sophisticated temporal and spatial blocking techniques.
- SSAM-based convolution is up to 2.5× faster than Nvidia's NPP library, and up to 1.5× faster than the highly-optimized ArrayFire library.
- We propose a novel Block-Register-Local-Transpose (BRLT) algorithm. To our knowledge, this is the first algorithm to apply the register cache method to compute SAT.
- We propose another efficient SAT computation algorithm that relies on an optimized intra-thread serial scan method to reduce inter-thread communication.
- Performance evaluation on the latest Tesla P100 and V100 GPUs demonstrates that SSAM-optimized SAT implementation outperforms the production-level OpenCV and Nvidia NPP libraries.

1.6 Dissertation Outline

The rest of dissertation is organized as follows. In Chapter 2, we introduce the overview of the background, e.g. Nvidia CUDA, Systolic array, GPU-oriented code automation, typical memory-bound kernels, etc. In Chapter 3, we describe the formulated software systolic array model (namely SSAM) and discuss mapping algorithms to SSAM. In Chapter 4, we map the 2D convolution algorithm in SSAM. In Chapter 5, we map the stencil algorithm in SSAM, including the spatial blocking and temporal blocking approaches. In Chapter 6, we map the Sumed Area Tables algorithm in SSAM. Chapter 7 shows some discussions, e.g. polyhedral analysis, code automation. In Chapter 8, we conclude this study. Chapter 9 discuss the future work. Note that we show the outline of this study as in Figure 1.2.

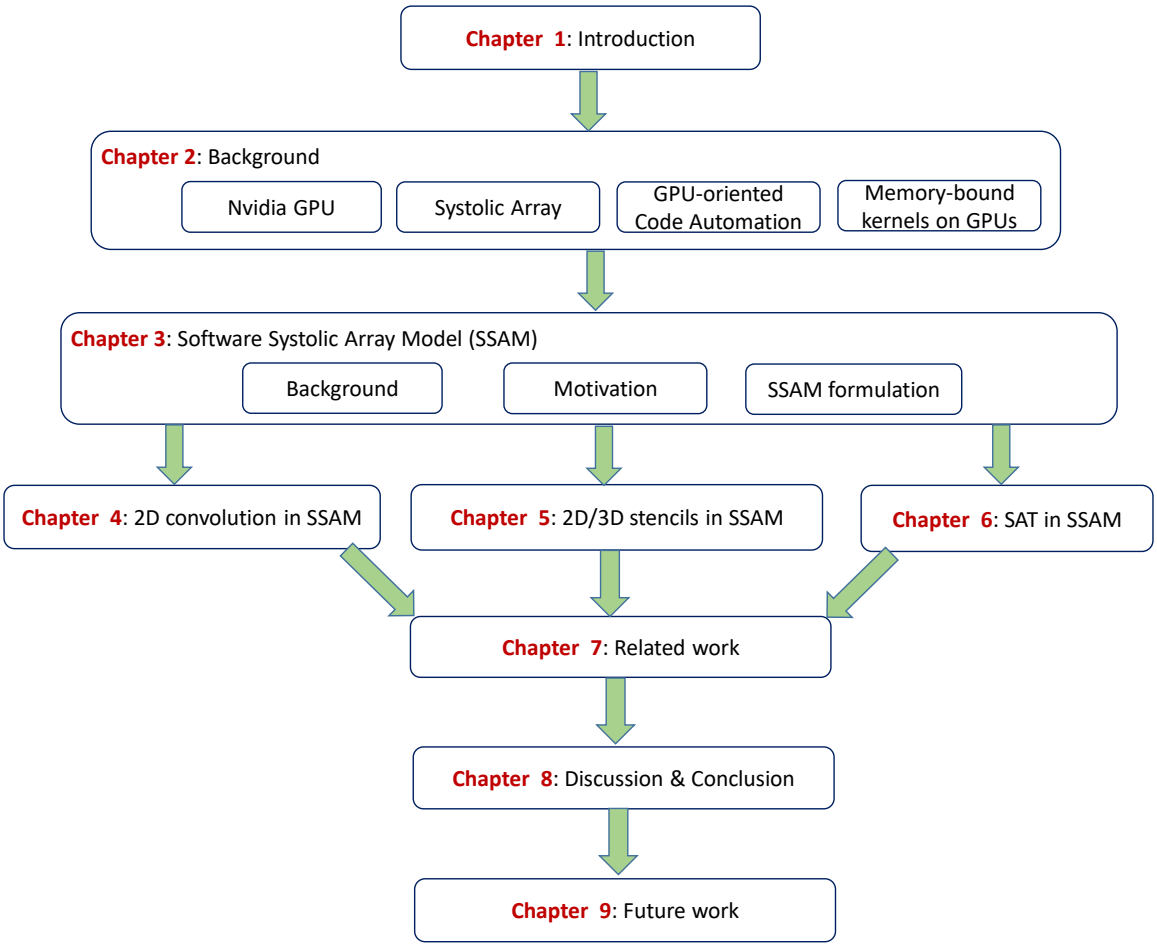


Fig. 1.2 Outline of this study.

Chapter 2

Background

In this chapter, firstly, introduce the concept of Nvidia GPUs the related Compute Unified Device Architecture (CUDA), e.g. memory hierarchy. Secondly, we illustrate the motivation of using registers as a cache to optimize applications. Thirdly, we list the shuffle intrinsics for efficient intra-warp communication. Fourthly, we revisit the hardware systolic array. Fifthly, we present the GPU-oriented code automation tools. Finally, we show the memory-bound kernels on GPUs.

2.1 Nvidia GPUs

In this section, we introduce Nvidia GPUs. A Graphics Processing Unit (GPU) is a programmable electronic circuit designed for display functions, e.g. accelerating the generation of images for the output display device. In contrast to Central Processing Units (CPUs), which are often employed for general computations and system control, GPUs are often dedicated to massive parallel applications. In other words, CPUs is latency-oriented processor and GPUs is a throughput-oriented processor. To maximize the overall throughput, GPU is designed by even sacrificing the sequential performance on many operations. More details can be found in the CUDA programming guide as in [30]. As Figure 2.1 shown, the computational performance (namely TFlop/s) increases dramatically year by year to meet the critical demanding for a large variety of fields, i.e. HPC, Artificial Intelligence (AI).

2.1.1 GPUs history

As introduced in [157], GPU is conventionally a specialized electronic circuit designed for rapidly (even realtime) creating images in a frame buffer for output to a display. This section describes the GPU history and the popularity of GPGPU.

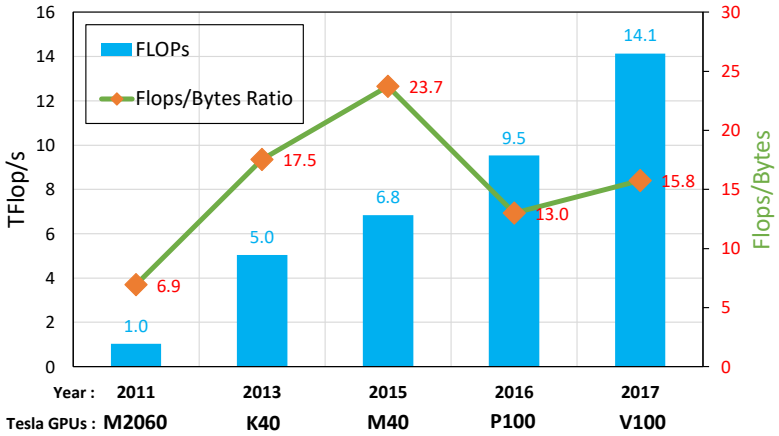


Fig. 2.1 GPU performance in single precision.

The commodity hardware is always designed to meet the specific requirements of its customers. As one of the most important hardware components in our computer systems, GPU was born (or developed) with the demanding of drawing high-quality images for video-game players. As early as the 1970s, a graph-specialized chip had been adopted in the arcade system boards as in [156]. At that time, it is mainly because the random-access memory (RAM) was too expensive to be utilized as the frame buffers for rendering the video games. The Fujitsu's MB14241 video shifter [160] was an exact example that was designed to speed up the rendering of sprite graphics for a variant of arcade games, e.g. Gun Fight, Sear Wolf, etc. In 1981, the monochrome and color display adapter, a particular computer component designed for displaying video, was adopted in IBM PCs. However, it is far from a modern GPU. In 1985, Array Technology Inc was created by three Hong Kong immigrants in Canada. it is soon renamed as ATI Technologies. This company was soon renamed as ATI Technologies and leads the market for many years with its graphics boards and chips. From early 1990, software-assisted 3D graphics were more and more popular in the arcade and computer games, which results in the high demand for hardware-optimized 3D graphics, i.e. Sega Model, Playstation video game consoles, etc. In the same period, OpenGL [165] interface emerged as a specified API for graphics programming. Noteworthy that early OpenGL was implemented by software, but soon improved by hardware implementation. Meanwhile, DirectX, a graphics API for Windows platform, was published for game developers. Driven by the requirement of realtime rendering from the market, the 3D accelerator cards start to evolve rapidly beyond the conventional rendering pipeline, resulting in borning the first genuine GPU, namely Nvidia Geforce 256 product. GeForce 256, a single-chip processor, is capable of processing at least ten million

polygons in a second. Nvidia in 2001 released the first GPU that supported programmable shading in the series of Geforce 3. Later in 2002, ATI announced Radeon 9700, the first Direct3D 9.0-supported GPU. Importantly, Radeon 9700 was programmable to implement both single-precision operations and loops with its pixel and vertex shaders. In the coming years, GPUs became more flexible for programming, while achieving orders of magnitude performance speedup. Nvidia GeForce 8800 proved to be more flexible for programming by integrating generic streaming processing units. The increased programmability and flexibility led to the popularity of GPGPU as nowadays.

2.1.2 GPUs for HPC

High-performance computing capability is always highly demanded, especially for lots of the emerging applications, e.g. scientific simulations, data mining, big data processing, high-resolution image reconstruction, real-time video processing, large scale machine learning, virtual reality, computational financing, etc. Conventionally, most of the applications are executed in CPUs (Central-Processing Units). However, the ever-growing demanding for compute capability was substantially out of the scalability of the performance of CPUs. Hence, a variant of accelerators is introduced, e.g. Nvidia GPUs (Graphics Processing Units), Intel Xeon Phi [25], FPGAs (Field Programmable Gate Arrays). Among these accelerators, GPUs become most popular to be used in a wide field, e.g. High-performance computing (HPC), Artificial Intelligence (AI), etc.

Originally, GPUs, a domain-specified device, is designed for rapidly rendering images from graphics. However, with the critical demanding of computing capability, GPUs gradually become programmable for accelerating the performance of common applications. Such kind of programming model is known as General-purpose computing on graphics processing units (termed as GPGPU). As the reported results for many CUDA-optimized applications [149, 69, 97], GPUs can achieve a magnitude of order faster than CPUs.

Despite GPUs achieves outstanding performance and demonstrates better performance scalability as illustrated in literature [30], the ever-growing demanding for the computing capability still enforces strong pressure over the performance scalability of GPUs. Adopting a completely divergent design principle, the GPUs, as a throughput-oriented accelerator, incorporates a much larger volume of streaming processors, that can launch thousands of light-weighted threads to hide the operation latency. In other words, GPU is different from the latency-oriented CPUs, which employs a large portion of the on-chip area for building the complex cache hierarchy, i.e. L1, L2, and L3.

2.1.3 Nvidia CUDA

Nvidia Compute Unified Device Architecture (CUDA) is a parallel computing platform and application programming model.

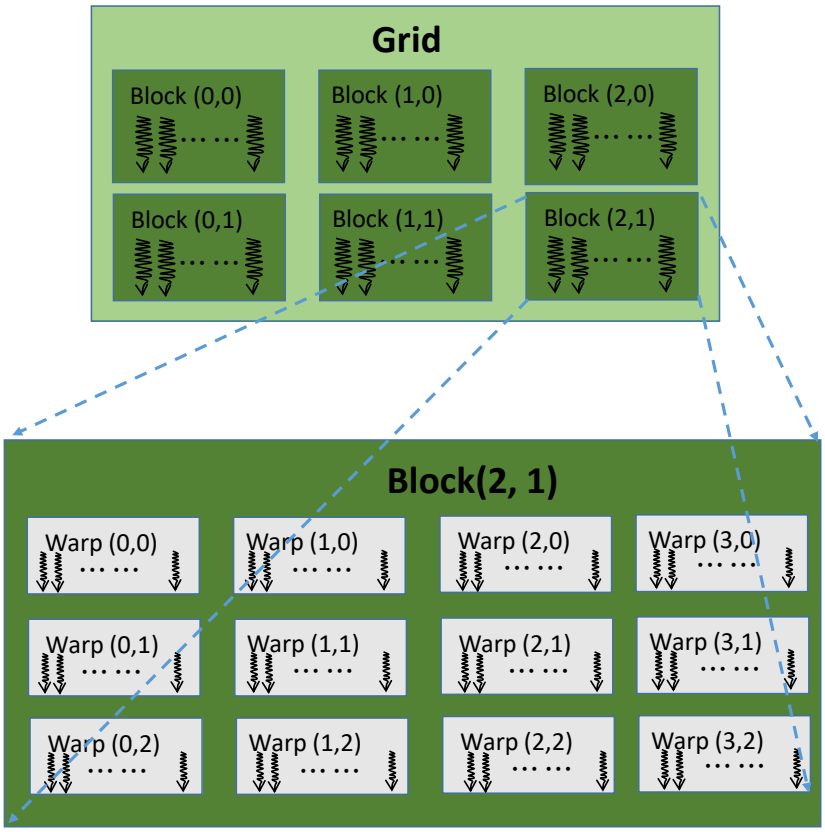


Fig. 2.2 CUDA execution hierachy.

CUDA execution hierachy In this section, we introduce the CUDA architecture. As Figure 2.3 shown, CUDA is built on an array of multi-threaded Streaming Processors (SMs). Massive thread-level parallelism is abstracted into a hierarchy of threads running in a single instruction multi-thread (SIMT) fashion. The threads in CUDA are grouped into warps, blocks, and grids. Thousands of threads are created, scheduled, and executed concurrently. Unlike the latency-optimized CPUs, GPUs are a throughput-optimized processor. More specifically, GPUs are capable of executing a massive number of threads concurrently and thus, can be seen as a coprocessor to the host device (or main CPU).

(I) **CUDA Warp**. In all generations of Nvidia GPUs, 32 threads execute in a group, called Warp. A warp executes the same instruction in lockstep. In other words, all threads in

a warp perform in a Single Instruction Multiple Data (SIMD) fashion. As a result, the divergence within a warp should be avoided. The total number of threads in a single Warp is defined as *WarpSize*[1].

(II) **CUDA lane** The CUDA lanes are referred to as the threads in a warp and many labeled by an index between 0 to *WarpSize-1*. In other words, each thread in a warp is assigned an ID, called *laneId*, which ranges from 0 to 31.

(III) **CUDA Block**. CUDA block is implemented by a cooperative thread array (CTA), which is a collection of concurrent threads executing the same CUDA kernel. A thread block is composed of multiple warps on the hardware side. In the latest GPUs, the maximum number of threads in a single block is fixed as 1024.

(IV) **CUDA Grid**. Thought the maximum number of threads that a block can contain is restricted, blocks can be batched together in a CUDA grid of blocks. Multiple blocks can execute concurrently (sequentially or in parallel), depending on the specified device and platform.

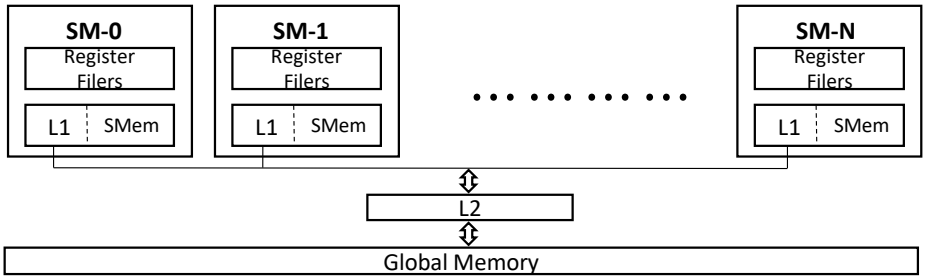


Fig. 2.3 GPU memory hierachy.

Table 2.1 GPU Memory Features

Memory Type	On/Off Chip	Cache	Access	Scope	Lifetime
Global	Off	L1/L2	Read/Write	per-thread	Thread
Constant	Off	N/A	Read only	kernel	host allocation
Shared	On	N/A	Read/Write	CTA	CTA
Registers	On	N/A	Read/Write	per-thread	Thread

CUDA memory hierarchy This section presents the CUDA memory hierachy. To approach the peak performance of GPU, it is essential to implement applications that efficiently utilize CUDA's memory hierarchy, i.e. global, shared, constant, and registers memory.

[1] WarpSize=32, through all generations of Nvidia GPUs

(I) **Global memory**. The largest off-chip memory with the highest R/W (Read/Write) latency. A coalesced access pattern is required for the CUDA kernels to achieve the highest bandwidth. As Figure 2.3 shown, the access to the global memory is via both L1 and L2 caches, which are fast on-chip memory with very limited capacities.

(II) **Shared memory**. A fast on-chip scratchpad memory, which shares space with the L1 cache as Figure 2.2 shows. The access scope is limited to a single CUDA block and thus, it is often used as a fast intra-block communication interface. Shared memory has 32 banks, which are organized as that successive 32-bit words map to successive banks. The bandwidth of each bank can achieve 32 bits per clock cycle. Note that shared memory has two addressing modes: 64-bit mode and 32-bit mode.

(III) **Constant memory**. As a special part of the global memory, constant memory is a fast off-chip and read-only memory, which is often used to store data that does not change while executing kernels. The capacity of constant memory is 64KB for all GPU generations. However, the constant memory is accessed via constant cache rather than L1/L2 (as will introduced in later paragraphs). Note that the specially designed constant cache is 8KB/10KB.

(IV) **Register files**. GPUs have a very large amount of register files and thus it is the motivation of building a virtual cache on the top of registers. Each SM has a set of 32-bit registers (as in Table 2.2) that are partitioned among all active warps, and data cache or scratchpad is partitioned among the CTAs (or blocks). In other words, registers are the fastest on-chip and thread-private memory. The access scope is restricted in a single thread.

(V) **L1 & L2 cache**. As Figure 2.3 shown, the global memory access that is cached in L1/L2 is performed with 128 bytes memory transactions. A cache line is 128 bytes and maps to a 128 bytes aligned segment in GPU memory. However, global memory accesses that are cached in L2 are only performed with 32-bytes memory transactions. Therefore, *over-fetch* can be reduced by caching in L2, e.g. the scattered memory accesses.

2.1.4 CUDA shuffle intrinsic

CUDA provides efficient intra-warp communication intrinsics, namely, shuffle. The purpose of shuffle intrinsics is used to read the specified registers which are held by other threads within the same warp. More specifically, without using the shared or global memory, threads

in a single warp can exchange registers with each other directly. In other words, the warp is a parallel unit that operates in lockstep as that all memory accesses and computations are performed in a SIMD fashion for all threads within a single warp. In terms of computing efficiency, using shuffle for in-register computation is superior in performance to the other memory types, due to its low latency and high throughput [22].

There are four different types of shuffle instrinsics as follows :

(I) **shfl_sync**. Directly copy data from the specified lane via laneId.

(II) **shfl_up_sync**. Copy data from the specified lane with lower laneId relative to the caller (or thread). *shfl_up_sync* can be substituted by *shfl_sync* as in Listing 2.1.

Listing 2.1 shuffle_up intrinsic

```
1 #define MASK 0xffffffff
2 #define shfl_up_sync(MASK, var) shfl_sync(MASK, var, laneId - var)
```

(III) **shfl_down_sync**. Copy data from the specified lane with higher laneId relative to the caller (or thread). Same to shfl_up_sync, the shfl_down_sync can be substituted as in Listing 2.2.

Listing 2.2 shuffle_down intrinsic

```
1 #define shfl_down_sync(MASK, var) shfl_sync(MASK, var, laneId + var)
```

(IV) **shfl_xor_sync**. Copy data from the specified lane based on bitwise XOR operation of own laneId. The typical usecase of this intrinsic is the FFT primitive as in literature [128].

In CUDA architecture, shuffle is a kind of intrinsic for Warp-wide communications. The more detail on other warp-wide intrinsic, e.g. __any, __all, __ballot, __brev, __clz, __ffs, __popc, can be found in literacture [30], since this study focuses on building a model by shuffle intrinsic.

2.2 Systolic Array

In this section, we illustrate the systolic array architecture and the typical applications. A systolic array is a network of processing elements (PEs) that rhythmically compute,

accumulate and transfer data through the system. The name of the systolic array is derived from drawing an analogy to how blood rhythmically flows through a biological heart as the data flows from memory in a rhythmic pattern passing through many elements before it comes back to memory. In other words, it is exactly a parallelized pipelining. Note that the mechanism of a PE shows in Figure 2.4, each PE is connected with the other PEs (as in Figure 1.1) and also has limited private storage. The systolic array was introduced in the 1970s and was successfully applied in CMU's processor (Warp-based computer and termed as iWarp [44, 125]) in 1990. There are several advantages of systeolic array :

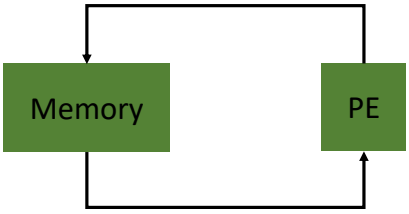


Fig. 2.4 A Processing Element (PE).

- The high degree of parallelism results in an extremely high throughput.
- Both data and control flow are very simple and regular.
- The architecture is highly robust and compact.

The disadvantages of systolic array are as follows:

- It is often hardwired for the specified problems.
- Inefficient to build a systolic array for the varied problems.

2.2.1 Systolic Array architecture

In a systolic array as Figure 1.1 shown, there are a set of identical yet simple Processing Elements (PEs), which can be hardwired for the specified computation (as will be introduced in Section 2.2.2). Typically, a systolic array consists of a large homogeneous network of primitive PEs, which can be hardwired or software configured for a specific computational task. In other words, the systolic array, a parallel computer architecture, is built as a monolithic network of tightly coupled data processing elements (or units). As introduced in literature [126, 20, 80, 81, 44], the systolic array architecture was invented by H. T. Kung and Charles Leiserson and successfully applied for many dense linear algebra computations

(banded matrices), e.g. solving systems of linear equations, matrix multiplication, LU decomposition. In the past decades, systolic arrays have been well researched. The authors in [77, 129, 169, 92, 23] have proposed well-structured systolic arrays architectures. A hardware systolic array is a computing pipeline network with physical interconnects between PEs. As Figure 1.1 shows, all of the PEs are connected with their neighbors. The PEs compute, store, and transfer their partial results to downstream neighbors in parallel. To sum up, we list some characteristics of a systolic array architecture as follows:

- A massive parallel and pipelined computing network
- Local communications between neighboring PEs
- Synchronous operations for both computation and communication
- Memory are reused multiple times
- A well-suitable architecture for implementation on VLSI and FPGA (as will be introduced in Section 2.2.2)

2.2.2 Systolic Array Application

This section presents the hardwired systolic array applications. To meet the critical demanding of high-performance for the domain-specified applications, the systolic array is the often adopted parallel computing architecture due to it is regular and easy to be implemented. A systolic array is ideally qualified for a specific computing purpose : e.g., matrix multiplication [76], convolution [78], DCT [89], data sorting tasks [37, 87, 141] ... etc. In other words, a systolic array is often hard-wired network [81] for the specific operations (i.e. multiply and accumulate) to perform massively parallel computations. The authors in [89] presented an efficient approach to design VLSI-based [2] discrete cosine transform (DCT) using the improved discrete Fourier transform (DFT) algorithm for the input sequence. In [124], the authors presented their implementations of one-dimensional and two-dimensional discrete wavelet transforms by systolic array architecture.

The systolic array is also employed in dynamic programming algorithms, e.g. DNA and protein sequence analysis [117]. Kung et al. [82] presented how to map real-time signal, image processing and scientific computing algorithms to the systolic arrays in VLSI [159] systems. Lam et al. [86] implemented an optimizing compiler on the Warp systolic computer (as introduced earlier) at CMU for high-performance computing. Iyengar et al. [60] presented a combined systolic array-content addressable memory architecture for real-time Gabor

[2] VLSI denotes Very Large Scale Integration.

decomposition, which is attractive for image compression due to the basis functions match the human visual profiles. The authors in [139] demonstrated a systolic array architecture for implementing a fast parallel decoding algorithm of one-point AG codes. To optimize the computation of Convolutional neural networks (CNNs) on FPGAs, Wei et al. [155] presented an automated solution to generate systolic array and achieved outstanding performance as that 461 GFlops for single precision and 1.2 Tops for 8$\sim$16 bits fixed point. The authors in [116] discuss several types of systolic array architecture with applications for motion analysis. Such pipelined architectures are designed for rapid motion detection, i.e. velocity, acceleration, and jerk.

2.3 GPU-oriented Code Automation

In this section, we introduce the main-stream code automation tools for GPUs. In application developing a workflow, code automation benefits to applications in many aspects, e.g. saving time, conserving resources, reducing errors and ensuring consistency. Generating code for GPUs is a convenient and inexpensive approach to improve the performance of applications due to that the programmer can take advantage of the massively parallelized architecture, namely GPUs, without knowing the details of the complex CUDA architecture and programming method. However, there often exists a significant gap between the achieved performance and the peak one in lots of the generated kernels, especially, for some memory-bound kernels. Hence, in this study, we focus on proposing a versatile execution model to improve the performance of such kinds of memory-bound kernels, allowing building a bridge between code automation and optimization. In the following sections, we introduce some state-of-the-art code automation tools for GPUs.

2.3.1 PPCG compiler

Verdoolaege et al. [151] presented a state-of-the-art Polyhedral Parallel Code Generation (termed as PPCG) for GPUs. PPCG is a general and robust source-to-source compiler based on polyhedral analysis techniques. More specifically, PPCG translates general C code to CUDA or OpenCL [145] accelerator code for GPUs. In PPCG, the affine transformations are extracted for data parallelism while using a code generator to orchestrate such parallelism over multiple levels of memory and parallelism. Constrained by the strict memory and concurrency, PPCG contributes to narrow the gap between parallelizing compilation for multi-core processors and code generators for many-core accelerators. Regarding the generated

codes on GPUs, PPCG can automate codes in both OpenCL [145] and CUDA interfaces. It is worth mentioning we focus on discussing its CUDA code in this study.

2.3.2 Halide

Halide is a programming language designed to generate high-performance image and array processing code on a variant of processors (e.g. CPUs, GPUs) and Platforms (e.g. Linux, Windows, etc). For simplification, Halide is a domain-specific language (DSL) designed to generates pipelines for image processing [111]. Halide is easy for programming since it is embedded in C++. More specifically, the C++ code that mixes an in-memory representation of a Halide pipeline can be compiled to an executable binary at the same time for the specified processor and platform. It is noteworthy that this study focuses on discussing the automated CUDA code that is generated by Halide.

2.4 Memory-bound kernels on GPUs

Table 2.2 Memory type and bandwidth of Tesla GPUs

Tesla GPU	Memory type	Bandwidth (GB/s)
M2060	GDDR5	148
K40	GDDR5	288
M40	GDDR5	288
P100	HBM2	732
V100	HBM2	894

In this section, we introduce the typical memory bound kernel on GPUs. Memory-bound kernels refer a class of computational problems and the required time of solving such kind of kernels on the specified processor is decided primarily by the memory access speed (or efficiency). As Figure 2.1 shown, the ratio of Flop/s and Bytes/s can be as high as 23.3. In real-world, the required performance for Flops/s is endless due to the large scale computational workload, it makes sense to provide as high performance as possible by the limited power consumption. However, few algorithms can achieve such a high rate of Flops/Bytes and thus, most of the computations are bounded by the memory access rather than the computation on GPUs. Though the P100 and V100 GPUs improve the balance of significantly by adopting the High Bandwidth Memory (HBM), which performs over 2.5 times faster than the DDR5-based memory as the Table 2.2 shows. Hence, the requirement for improving the performance of memory-bound kernels is a prevalent motif. In this study,

we focus on a collection of a memory-bound kernel with regular memory access, e.g. Scan operator, convolution, stencils, and Summed Area Table. Furthermore, we map algorithms to a virtual systolic array which is built on the top of CUDA architecture. All detailed illustrations will be presented in the later sections.

Chapter 3

Software Systolic Array Execution Model

This chapter firstly presents the background and motivation of building a virtual systolic on the top of CUDA architecture. Then, we formulate a virtual systolic array, termed as Software Systolic Array Execution Model (SSAM). Finally, using two illustrative examples, we show the details of expressing algorithms in SSAM.

3.1 Background & Motivation

This section illustrates the background of CUDA optimization and the motivation for building a general execution model. To achieve the peak performance of GPUs, understanding the detailed CUDA architecture and utilizing the complex memory hierarchy as introduced in earlier sections are of fundamental and critical for optimizing CUDA applications. More specifically, several considerations are significant in achieving high performance on GPUs : (1) Coalesced access to global memory; (2) Reuse of data in the fast on-chip memory, i.e. registers, scratchpad; (3) Sufficient concurrency to meet the requirement of physical cores; (4) Effective use of limited registers and shared memory. Note that their capacities are listed in Table 1.1.

3.1.1 Bandwidth optimization

As discussed in literature [58, 49], Amdahl's law is often be used to predict the theoretical maximum improvement possibility by optimizing a particular part of a parallel computing system that uses multiple processors. Indeed, it answers the question that what should be optimized or parallelized for the CUDA-optimized applications? Regarding the prevalent

memory-bound kernels on GPUs, reducing the global memory access and increasing the data reuse may significantly improve the performance of such kernels on GPUs due to the improved memory bandwidth. Theoretically, this effection can be justified by the Roofline model [163] easily; The strategies of optimizing memory memory access on GPUs, i.e. spatial blocking (as in Section 5.1.3), temporal blocking (as in 5.1.4), register cache [85, 41, 59, 8], are fundamentally adopted in a variant of applications.

3.1.2 Locality optimization

To fully take advantage of registers to perform the efficient in-register computations at high throughput and low latency, we build a virtual cache on the top of register files (termed as register cache). Note that the concept of register cache can also be introduced in literature [85, 41, 59, 8]. More specifically, register cache, a software abstraction, is implemented by CUDA shuffle intrinsic. Regarding kernels with regular memory access, the cached data often been reused multiple times and thus, the global memory access can be reduced. It results in performance improvement in two aspects. One is the higher memory bandwidth available, the other is that we can perform more in-register computation rather than using scratchpad or global memory.

3.2 SSAM: Software Systolic Array Model

In this section, we propose an execution model, named Software Systolic Array Execution Model (SSAM), that enhances the performance of regular memory-bound kernels on GPUs. SSAM is a generic model that could be utilized by many applications. we first introduce a formulation of systolic arrays. Then, we discuss how our CUDA implementation leverages the device features to realize SSAM. Next, we discuss a formal method for how specific algorithms can be expressed in this model. Finally, we elaborate on how the convolution and scan operator [26, 53] use cases are expressed in SSAM.

Our work is motivated by the recent revival of systolic arrays and similar architectures such as Tensor cores of Nvidia GPUs, Intel TBB [137] data flow graphs, and wavefront accelerators (mainly FPGAs [70] and ASICs [12]). Tensor cores are specially optimized for matrix multiplication, however, SSAM is a general model that is applicable to a wide range of algorithms.

3.2.1 A Formulated Processing Element in Systolic Array

A systolic array [79] is a structured computing model composed of many interconnected, yet independent, Processing Elements (PEs) as shown in Figure 1.1. In parallel, all PEs compute the partial results for a specified function, store results locally, and then send them to neighbor PEs for the next cycle. The computational and data storage behavior of each PE can be formulated as

$$s \leftarrow ctrl(r \otimes x) \oplus s \tag{3.1}$$

where s is a partial result held by each PE, r is an external coefficient (e.g. for convolution r is the weights), and x is an input value. Both $\otimes$ and $\oplus$ are basic arithmetic operations, and $ctrl(E)$ is a control function having the value of 0 or E, i.e. the output of $ctrl(E)$ is 0 or E.

In addition, as illustrated in Section 2.2, the variable of s in Equation 3.1 performs the function of memory in each PE as in Figure 2.4.

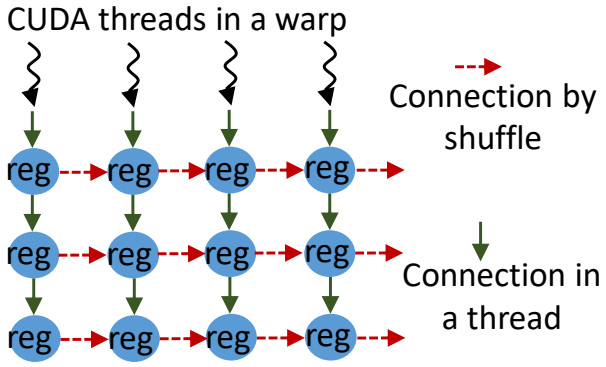


Fig. 3.1 Software Systolic Array on CUDA.

3.2.2 SSAM: Software Systolic Array Model

SSAM is built as an execution model to express a variety of algorithms, with regular access patterns, in a systolic array fashion. In other words, it stimulates the mechanism of hardware systolic arrays in CUDA architecture. Our target in this paper is to improve the performance of memory-bound kernels with regular memory access patterns. However, SSAM, in general, is not limited to memory-bound kernels and could be extended to compute-bound kernels, such as GEMM. In this section, we present SSAM's formulation and discuss motivating examples. Three core techniques contribute to the SSAM model:

- Efficient in-register computation via register cache;

- Fast intra-warp communication via shuffle instructions;
- Parallel accumulation of partial sums.

We build a software systolic array on the top of a virtual register cache that is used for efficient data access and reuse. In addition, the shuffle instructions are used for low latency communication between threads in a warp. As Figure 3.1 shows, each register performs the same function as a PE in Figure 1.1. It is important to iterate that the registers (PEs) in the vertical direction belong to the same CUDA thread, while the registers in the horizontal direction belong to different threads.

In this paragraph, we describe more details of the Figure 3.1. The *reg* is a register and the *arrows* mean connections. In the vertical direction, registers are in the same thread. In the horizontal direction, registers are exchanged by the shuffle instruction. In other words, similar to the hardwired systolic array in Figure 1.1 like that: in the vertical direction, the registers are connected with their neighboring elements naturally; in the horizontal direction, the registers are connected with their neighboring elements by shuffle intrinsic. Note that all registers are employed to simultaneously compute, store, and transfer their partial results to the downstream neighbors.

The similarity between systolic array PEs and our registers can be listed as follows: (i) the ability to execute any arithmetic operations by oneself (ii) the ability to store partial result by oneself (iii) the ability to pass the partial result to neighbors (direct register access in the vertical direction, and using shuffle in the horizontal direction), (iv) most importantly, all of the operations are performed simultaneously.

It is worth mentioning that the use of in-register computing and partial sums have been proposed in specific implementations of individual algorithms (as will be discussed in the related work in Section 7). On the other hand, the model proposed in this paper is a robust and versatile model that can be leveraged by a wide variety of problems with different patterns of data dependency, as will be demonstrated in the following sections.

3.2.3 What is SSAM?

As explained earlier, SSAM is built as a software systolic array on the top of CUDA architecture. SSAM mimics the behavior of hardwired systolic by CUDA warp, register cache, and partial sums as Figure 3.2 shows. Since the use of register cache, the partial sums can be computed efficiently, e.g. loading data from register cache, storing data to register cache. Except for partial sums computation, partial sums operation includes partial transfer, which mimics the PEs communicate with their neighbors as in Figure 1.1.

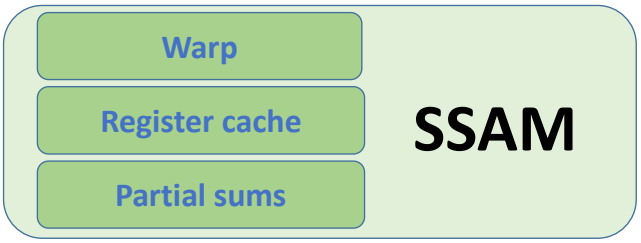


Fig. 3.2 SSAM.

For simplification, a CUDA warp can be seen as a small systolic array (or SSAM) and thus, CUDA is made of several such kinds of soft systolic arrays due to CUDA can launches thousands of warps concurrently. Since SSAM is implemented by software, the inter-SSAM communications are required to be performed via shared memory or global memory. However, intra-SSAM communication can be performed by efficient shuffle intrinsic as Figure 3.1 shows. Comparing with hardware systolic array, SSAM is more flexible for programming. SSAM can perform all of the arithmetic operations, e.g. sqrt, mod, etc. In addition, we can freely perform intra-SSAM communications that are different from the hardware systolic since the communication patterns in hardware systolic array are fixed in advance. It is noteworthy that the complex communication patterns in SSAM can be found in Figure 3.3 (scan operator) and chapter 6 (SAT computation).

3.2.4 Expressing Algorithms in SSAM

In order to express algorithms in SSAM, we build an algorithmic formulation that extends the prior work in the literature [109]. From the perspective of a CUDA warp, an algorithm can be formulated as a four-tuple

$$J = (X, Y, O, D) \tag{3.2}$$

where

- X : input variables of J
- Y : output variables of J
- O : computing operations of J
- D : dependencies of J

In the model J, the register cache is used for storing X and Y (more details about register cache will be discussed in Section 4.3.2), their ranges of values in J are R. In contrast to hardware systolic arrays, SSAM can perform computing operations O like Equation 3.1,

i.e. all arithmetic and intrinsic operations supported by CUDA. In addition, the graph representing the dependencies D can take any shape and is not limited to a structured mesh. This is due to the fact that the shuffle operation allows arbitrary threads in a warp to exchange values held in registers. It is worth mentioning that the partial results are updated as specified in Equation 3.1, and are passed according to the dependency graph D. There are several methods to extract the dependency graph D. We briefly introduce one of those methods in Section 8.1.1 : using the polyhedral model [43] to represent D.

3.2.5 Motivating Example 1: 1D Convolution

We use 1D convolution as a motivating example to illustrate how an algorithm can be expressed in SSAM. Given an input array $A = [a_0, \cdots, a_{n-1}]$ and filter $F = [f_0, f_1, f_2]$, the output of convolution is an array $B = [b_0, \cdots, b_{n-1}]$, such that $b_i = \sum_{z=i-1}^{i+1} a_z \cdot f_z$. We use registers to cache the input data, with one thread computing one element. In SSAM, we map the 1D convolution to a warp (Equation 3.2) such that the input $X = [a_k, \cdots, a_{k+WarpSize-1}]$ and output $Y = [b_k, \cdots, b_{k+WarpSize-1}]$, where $0 \leq k \leq n - WarpSize$. For the computation O (in Equation 3.1) where r$\in$F, $\otimes$ and $\oplus$ are multiplication and addition operations, respectively. The dependency graph D can be represented as in Figure 4.4, where the $ctrl() \equiv 1$ is fixed.

The 1D convolution is simple to some extent. Furthermore, the complex case of **2D convolution (or 2D stencil)**, is proven to benefit of a systolic model optimization similar to SSAM, more details can be found in the later chapter 4.

3.2.6 Motivating Example 2: Scan Operator

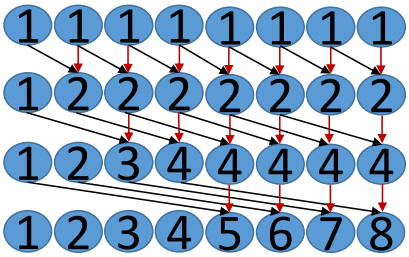


Fig. 3.3 Eight elements KS-scan [72].

Scan operator is the sums of prefixes (running total) of an input sequence [72, 53]. Even though we do not evaluate Scan operator in this paper, we choose Scan operator as an example of mapping algorithms in SSAM to demonstrate its versatility. Given an input array $[a_0, \cdots, a_{n-1}]$, the output of Scan is an array $[b_0, \cdots, b_{n-1}]$, such that $b_i = \sum_0^i a_i$. For simplicity, we use registers to cache the input data where each thread holds one element. In our model,

we map the Scan operation to a warp in SSAM such that the input $X = [a_k, \cdots, a_{k+WarpSize-1}]$, and output $Y = [b_k, \cdots, b_{k+WarpSize-1}]$, where $0 \leq k \leq n - WarpSize$. As to the computation operations O, in each computing stage (Equation 3.1), $r \equiv 1$, $\otimes$ is a multiplication and $\oplus$ is an addition operation. Since we use the Kogge-Stone Scan algorithm [72], the dependency graph D is similar to Figure 3.3, namely the arrows in Figure 3.3 can be expressed as *ctrl()* in Equation 3.1. Note that the algorithmic computation of several parallel scan operators can be found in Section 6.2.1.

A one-dimensional scan operator is simple to some extent. Furthermore, the complex case of **two-dimensional scan**, known as Summed Area Tables (SAT), is proven to benefit of a systolic model optimization similar to SSAM, we give a detailed illustration in chapter 6.

Chapter 4

Two-dimensional Convolution in SSAM

In this chapter, we present the SSAM-based 2D convolution implementation. In Section 4.1, we overview the background of 2D convolution. In Section 4.2, we illustrate the motivation. In Section 4.3, we present the detailed implementation. In Section 4.4, we build a performance model for our implementation. In Section 4.5, we demonstrate the achieved performance. In Section 4.6, we conclude.

4.1 Background

Convolution-based operators, namely filtering, are of fundamental importance to a vast range of digital signal and image processing applications. For example, the Gaussian-filter and Laplace-filter are widely used to enhance image quality as Figure 4.1 shows. More specifically, Difference of Gaussian (DOG) is commonly applied for Scale-Invariant Feature Transform (SIFT) [96] to detect robust local-features in images. In the domain of artificial intelligence, convolution is also a core technology of Deep Neural Networks (DNN). Using multiple convolutions to extracting sparse features from the input matrix, Convolutional Neural Networks (CNN) [75] achieves outstanding performance in many challenging tasks, e.g. text indexing, voice recognition, and image classification.

4.2 Motivation

As a typical memory-bound algorithm, to the author's knowledge, convolutions often become the bottleneck of the computation in many applications. Hence, it is significant to improve the performance for convolution to meet low latency demand. Many effective ways have been introduced to speed up the 2D convolution computations. FFT-based approaches [150] [73]

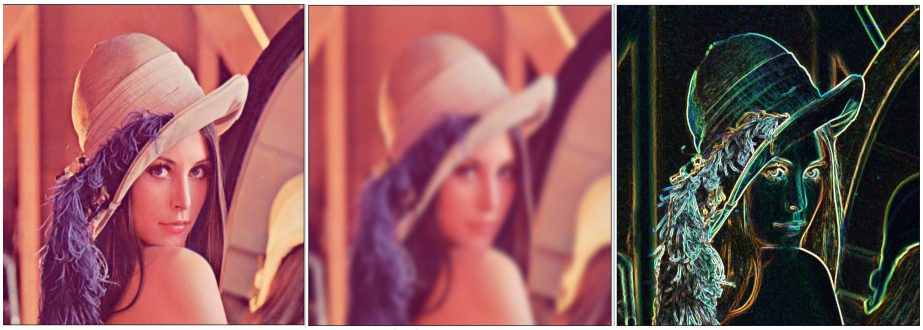


Fig. 4.1 Image processing by 2D convolution.

reduce the computation complexity from O(N^4) to O($N^2 \log^2 N$) regardless of the variance in filter sizes; it is particularly suitable for convolutions with larger filters. As for some filters with special sizes, Winograd's algorithm [88] can also reduce the computation to some extent. Another popular approach is that of unrolling the input data and filter coefficients to two matrices [21], which are stored by the workspace buffer, then applying the highly optimized General Matrix-Matrix Multiplication (GEMM) [67] function to accelerate the convolution computation. All of those methods have been incorporated as parts of the *cuDNN*'s [24] convolution functions.

GPU is suitable for optimizing the 2D convolution in many aspects. For computational efficiency, GPUs provide a huge computing capability over conventional CPUs as mentioned in earlier chapters, and have been successfully used in High-Performance Computing (HPC) [142] applications. Also, GPUs are widely used to accelerate Deep Learning (DL) workloads [11] [61]. Hence, GPUs, especially CUDA-enabled GPUs, are well-suited many-core accelerators for computing convolutions with high throughput.

According to the memory hierarchy of CUDA architecture [28], caching data by explicitly managed scratch-pad memory is effective in hiding the latency of accessing global memory. Naturally, this technology is widely used to optimize the application's performance (including convolution) by many state-of-the-art libraries, such as ArrayFire [168].

We use a 2D convolution algorithm, a typical memory-bound algorithm, as a detailed example to explain how to implement a systolic array-liked kernel. namely, we propose an algorithm that directly computes convolution without relying on scratchpad or cache memory. More specifically, we manually cache data by register files directly, accumulate the partial correlation results, and communicate the partial results via shifting register files with CUDA's *shuffle_up* instruction. Results show that our algorithm achieves the highest efficiency for computing 2D convolution, compared to existing state-of-the-art libraries.

4.2.1 2D Convolution definition

Mathematically, convolution combines two linear functions to form a third one to measure the correlation of overlap between functions. As Figure 4.1 shown, convolution is often used for image processing, e.g. smoothing, denoising, sharpening, and edge detection. The canonical form of 2D convolution is

$$(f * w)(x,y) = \sum_{t=c}^{d} \sum_{s=a}^{b} f(x-s, y-t) \cdot w(s,t)$$

Where $*$ and $\cdot$ are convolution and multiplication operators, respectively. f(x, y) denotes a 2D matrix, while w is a filter of size (M, N) applied to the image, where M=b-a+1, N=d-c+1. Hereafter, we assume that the size of the 2D matrix is (W, H) where W is the image width and H is its height.

Note that throughout this chapter, we use W, H, M, N as defined in this section.

4.3 Implementation of 2D convolution in SSAM

Detailed techniques required in implementing the SSAM model are illustrated in this section, such as coalescing global memory accesses, caching data by register files, computing and transferring partial sums, and overlapped blocking. Using 2D convolution as a motivating example, we discuss the implementation of SSAM. Additionally, we mention the special considerations given to stencil operators when necessary.

4.3.1 Mapping 2D Convolution to SSAM

Using the SSAM model introduced in the previous section, we implement an efficient 2D convolution algorithm. Regarding the mapping of algorithms to SSAM as J (Equation 3.2), the input X and output Y are addressed in Section 4.3.2. The computing operation O is discussed in Section 4.3.3, and the dependency graph D is discussed in Section 4.3.4. The detailed 2D convolution implemented by SSAM model is shown in Listing 4.1:

(i) All of the filter weights are stored into shared memory (lines 7∼12).

(ii) A subset of the image data residing in global memory is cached into registers (lines 13∼14). Since registers are a limited resource, the register cache is managed with careful consideration. For this purpose, we introduce a sliding window scheme (more details on that in the next section).

Listing 4.1 SSAM-based 2D Convolution: CUDA kernel. M, N, W, H are defined in Section 4.2.1, P, B and C are illustrated in Section 4.3.2.

```
1  template<typename T, int B, int P, int M, int N>
2  __global__ void 2Dconvolution(const T* src,
3      T* dst, int W, int H, const T* weight) {
4     const int C = P + N - 1;
5      //register files
6     T data[C];
7     __shared__ T smem[N][M];
8     T* psmem = &smem[0][0];
9      //1, Load filter weights to shared memory
10    for (int i=threadIdx.x; i < N*M; i += B)
11       psmem[threadIdx.x] = weight[threadIdx.x];
12    __syncthreads();
13    //2, Load data from global memory to registers
14     data[...] = src[...];
15    //3, Compute convolution via registers
16    #pragma unroll
17    for (int i=0; i<P; i++) {
18       T sum = 0;
19       #pragma unroll
20       for (int m = 0; m < M; m++) {
21          if (m > 0)
22              sum = __shfl_up_sync(0xffffffff, sum, 1);
23          #pragma unroll
24          for (int n = 0; n < N; n++) {
25              sum = MAD(data[i + n], smem[n][m], sum);
26          }
27       }
28       data[i] = sum;
29    }
30     //4, Store Result to Global Memory
31     dst[...] = data[...];
32 }
```

(iii) As shown in Figure 4.2, according to the sliding window[1] position and filter height, we fetch both sub-vector v and w from the register cache and filter coefficients, respectively. Next, we compute the partial sums by the fused multiply-add operator MAD (lines 24∼26) and transfer the partial sums to the neighbor threads via the *shuffle* primitive (line 22).

(iv) Repeat step (iii) M times for all of the sub-vectors $(w_1, \ldots, w_M)$, then store the final partial sums to the register cache again (line 28).

[1]We use a sliding window to compute several output points per thread. This method improves both data reuse and ILP

(v) We move the sliding window step by step for a total of P times (line 17) in Listing. 4.1. At each step, we repeat the convolution computation, namely (iii) and (iv).

(vi) Finally, the convolution results, which reside in the register cache, are stored back to global memory (lines 30∼31).

The following sections elaborate on the steps of the algorithm.

4.3.2 Register Cache & Coalesced Memory Access

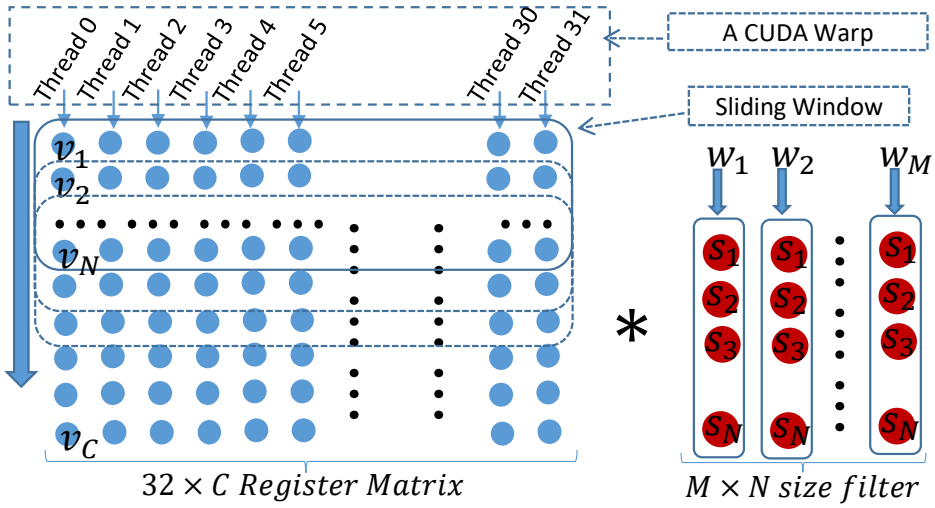


Fig. 4.2 Register Cache.

In Figure 4.2, we illustrate computing 2D convolution by a single warp to 32×C register-cached words and M×N filter matrix.

In SSAM and CUDA best practice in general, it is crucial for performance to account for coalesced global memory access. Hence, we make sure that all of the threads in a warp read data from global memory contiguously (one element per thread) as shown in Listing 4.1. The operation is repeated to cache multiple lines of data from global memory into register files, line by line. As illustrated in Figure 4.2, each thread in a single warp caches C elements as

$$C = N + P - 1 \tag{4.1}$$

Where N is the filter size. Each thread computes the output of P elements using a sliding window. The sliding window is designed such that a portion of the data in the register cache can be reused when computing the neighboring output points. More specifically, the computation of the convolution of point P in a thread can reuse the data in the register cache

loaded when computing the convolution of point *P-1*. Using this scheme, at any given point, a WarpSize×C register matrix is stored in the register cache.

In Figure 4.2, the left side of the figure illustrates how to populate the register cache for a warp. In a single warp, each thread reserves C registers for storing data. The register cache size, and systolic array size, in each warp is equal to WarpSize×C. The right side of the figure shows how to cache the filter matrix. We store the filter coefficients in shared memory, then compute the convolution by moving the sliding window step by step P times. At each step we compute the inner products of $[v_i, v_{i+1}, ..., v_{i+N-1}]$ with $w_1, w_2, ..., w_M$ as shown in Fig 4.3. Next, we shift the partial inner product to the neighboring threads as shown in Figure 4.4. As introduced in earlier section, the total number of threads executed together in a working unit (i.e. CUDA warp) is relatively small. Hence, this enforces a limit on the systolic array size and parallelism. Fortunately, this challenge is tackled by such a sliding window technique that is simple yet effective.

4.3.3 Computing Partial Sums

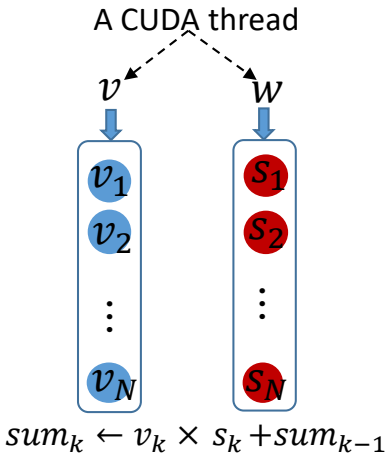


Fig. 4.3 Computing Partial Sums.

In Fig 4.3, V and S are register data and filter coefficients, respectively. Fig 4.3 illustrates an example of computing partial sums in parallel.

In this section, we describe the computation of partial sums. Computing the partial sum for 2D convolution by Equation 3.1 is similar to the 1-D convolution example in Section 3.2.5. More specifically as seen in Figure 4.3, all threads simultaneously, in a systolic fashion, compute the partial sum (sum_k) between a register vector v ($[v_i, v_{i+1}, ..., v_{i+N-1}]$) and a column of filter $w_1, w_2, ...,$ or w_M. Additionally, the vector v is held by each thread and w is managed by shared memory. It is necessary to access the filter weights in the same order as

the data is stored in register cache: namely, unit-strided access in the vertical direction as shown in Figure 4.3. The partial sum requires N multiplications and N-1 addition operations. The multiplication and addition operations are typically optimized to fused-multiply-add (MAD instruction in CUDA [30]). For the M×N filter, the inner products are computed M times (as shown in Figure 4.2) to process a single output element of convolution (Listing 4.1 line 20∼27).

4.3.4 Transferring & Accumulating Partial Sums

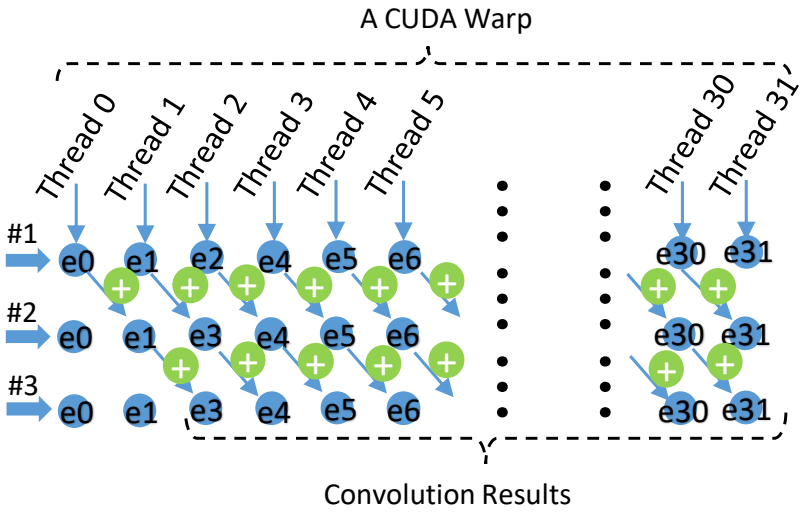


Fig. 4.4 Transfer Partial Sums in parallel.

In Fig 4.4, e_i is a partial result computed by thread i. Threads (0∼31) represent all threads in a warp. The indicator #i points to the i^{th} time of shifting partial sums in a warp. $\oplus$ is an addition operator.

In the SSAM model, all of the threads within a single warp accumulate and transfer the partial sums. We use the shuffle instructions to transfer the partial results to the downstream neighbor threads for further accumulations. In Figure 4.2 the M×N filter is decomposed into M vectors, namely w_1, w_2, ..., w_M. Each partial sum is computed between register vector v ($[v_i, v_{i+1}, ..., v_{i+N-1}]$) and filter vector w. Then, all of the inner products, namely partial sums, are shifted to the right side neighbor thread within a single warp using the CUDA *shuffle_up* function. In Figure 4.4, all of the registers are shifted only once at each step (Listing 4.1 line 22). Next, the shifted partial results are added to the accumulated results by each thread (Listing 4.1 line 25). This process is repeated *M-1* times (Listing 4.1 line 20∼27). Finally a row of convolution results could be attained from a group of threads,

namely whose *laneIds* [30] ranges from *M-1* to *WarpSize-1*. By moving the sliding window once, each warp of threads computes *WarpSize-M+1* convolution results.

4.3.5 Overlapped Blocking Scheme

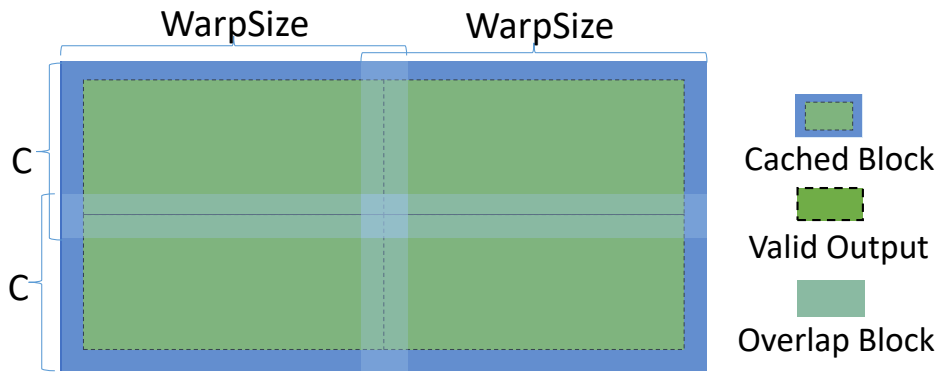


Fig. 4.5 Overlapped Blocking.

In Figure 4.5, the cached block size is WarpSize×C, the valid output size is (WarpSize-M+1)×P. Overlapped block size is a function of the specified filter size.

The overlapped blocking scheme is widely adopted to improve the concurrency in time-tiled stencils computations [174, 74]. We use this scheme to eliminate warp-divergence, which can have negative performance effects. As Figure 4.5 shows, we design our overlapped blocking algorithm to avoid intra-block communications that may cause branching when computing the partial sums and shifting of the register. Using multiple loops for computation is the heaviest and most complex part of computing the 2D convolution (Listing 4.1 lines 16∼27). The overlapped blocking scheme enables all of the threads in our CUDA kernel to perfectly execute without any branching. Halo layer(s) overhead is introduced when using overlapped blocking due to that the shuffle primitives work only for a single warp: a redundancy method for halo layers, along with a redundancy analysis, is necessary. We provide an elaborate analysis of the halo layers impact on performance (performance model in Section 4.4.3).

4.3.6 Caching Filter Coefficients

We cache filter weights by shared memory. Filter weights are often tens of bytes (e.g. 36B for 3×3 and 100B for 5×5). Since the weights are shared by all of the threads in a CUDA block, it is reasonable to access the weights via shared memory. It is also possible to use other kinds of memory, such as constant memory and texture memory. A small number of weights

could also be passed to the CUDA kernel directly as arguments. However, considering our commitment to scaling to large filter sizes, we use shared memory in our implementation. Our algorithm performs a broadcast read pattern to the shared memory: all threads in a CUDA block access the same address of shared memory. This assures no bank conflict problems, as described by CUDA guide [30].

4.3.7 Number of Required CUDA Blocks

We use a one-dimensional block in our implementation, BlockDim=(B, 1, 1). *B* is equal to blockDim.x, both blockDim.y, and blockDim.z are 1. In each CUDA block, its warp count is defined as *WarpCount = B/WarpSize*. The required CUDA grid dimensions are expressed as

$$GridDim.x = \lceil \frac{W}{WarpCount * (WarpSize - M + 1)} \rceil$$

$$GridDim.y = \lceil \frac{H}{P} \rceil = \lceil \frac{H}{C - N + 1} \rceil$$

$$GridDim.z = 1$$

4.4 Performance Model

In this section, we introduce a performance model to analyze the effectiveness of SSAM over the conventional scratchpad memory implementations. Also, we analyze the overhead of processing the halo area required in the overlapped blocking scheme (in Section 4.3.5). For most of the memory-bound regular applications that could be implemented with SSAM, the analysis results should be valid, yet the details of the analysis may vary from case to case. We use 2D convolution as a motivating example in the following discussions.

4.4.1 Micro-benchmarking

We rely on micro-benchmarking to better understand the characteristics of GPUs. Previous work investigated some details about memory latencies and throughput of GPUs [164, 104]. We further extended and improved the the work in [164] (known as *cudabmk*) to measure the shuffle instruction and shared memory access latencies. As Listing 4.2 shown, we implement a extended cuda kernel for estimating the latency of shuffle instruction. In line 6, $REPEAT\_N(x)$ is a macro, which continuously copies its variable (namely x) N times, and N is set as 256 in our measurements. To avoid dead code elimination to line 6, we use a

Listing 4.2 CUDA Kernel for estimating __shfl_up latency

```
1  start_time = stop_time = 0;
2  for (;start_time >= stop_time;){
3      for (int i=0;i<iterative;i++){
4          __syncthreads();
5          start_time = clock();
6          REPEAT_N(t2=__shfl_up(t1); t1=__shfl_up(t2);)
7          stop_time = clock();
8      }
9  }
10 out[0] = (unsigned int )(t1 + t2);
11 result[...] = start_time;
12 result[...] = stop_time;
```

dummy output (*out* variable in Listing 4.2) as a trick to the compiler in line 10. At the end of the kernel, the recorded execution time (namely $start\_time$ and $stop\_time$) are returned to host side. Finally, the average execution time of shuffle instruction can be calculated as

$$AvgTime = \frac{stop\_time - start\_time}{2 * N} \tag{4.2}$$

since N is sufficiently large, the overhead of calling clock() function can be eliminated by averaging as Equation 4.2. In our evaluation, the value of $iterative$ (line 3) is set as 2 and, thus only the last iteration result is used as output. Since shuffle instruction is a warp operator, for its latency evaluation only one block and one warp are launched for execution.

The measured latency values are listed in Table 4.1, All of the latency values are measured by our micro-benchmarks in the unit of cycles/warp. T_{smem_read} is the latency of reading shared memory As reported by Nvidia's official programming manual [121], the throughput of shuffle, addition and Boolean *AND* operations are 32, 64 and 64 operations/clock, respectively.

4.4.2 Efficient In-register Partial Sums Computation

In this section, we demonstrate the benefits of using SSAM by comparing the latency of computing a single output element of 2D convolution using SSAM versus the conventional shared memory implementation.

We define the following parameters for modeling the compute time. T_{smem_read} is the latency of reading shared memory, T_{reg} is the latency of reading/writing register, T_{shfl} is the latency of shuffle instruction, T_{mad} is the latency of MAD operation, T_{gmem_read} and T_{gmem_write} are respectively the latency of reading and writing global memory. Suppose M×N filter

Table 4.1 The latency of different operations.

GPU	Operation	Latency (clocks/warp)
Tesla P100	shfl_up_sync	33
	shfl_down_sync	32
	shfl_sync	31
	ADD, SUB, MAD, AND	6
	T_{smem_read}	33
Tesla V100	shfl_up_sync	22
	shfl_down_sync	22
	shfl_sync	22
	ADD, SUB, MAD, AND	4
	T_{smem_read}	27

coefficients are cached in shared memory in advance. Conventionally, the image data is cached in shared memory, the latency of computing an output element is

$$L_{smem} = M \cdot N \cdot (T_{mad} + 2 \cdot T_{smem\_read} + 2 \cdot T_{reg})$$

Where $2 \cdot T_{smem\_read}$ is the time to perform one load of a filter weight plus one store of cached data from shared memory to registers. In the case of SSAM, the image data is cached in register cache, the latency may be expressed as

$$L_{reg} = M \cdot N \cdot (T_{mad} + T_{smem\_read} + 2 \cdot T_{reg}) + (M-1) \cdot T_{shfl} \tag{4.3}$$

It is worth mentioning that we require (M-1) shuffle instructions for intra-warp communication: no shared memory is required for intra-warp communication. The difference of computing an output element between the shared memory method and register cache can be derived as

$$\begin{aligned} Dif_{smem\_reg} &= L_{smem} - L_{reg} \\ &= M \cdot N \cdot T_{smem\_read} - (M-1) \cdot T_{shfl} \end{aligned} \tag{4.4}$$

Based on the metrics in Table 4.1, The result is $Dif_{smem\_reg} \gg 0$, where $M \geq 2$ and $N \geq 2$.

In conclusion, the register-based partial sums computation method is more efficient than the shared memory method.

4.4.3 Halo Layer(s) Overhead

In this section, we analyze the benefits and overhead of using overlapped blocking (as shown in Section 4.3.5) by comparing the shared-memory-cache implementation with SSAM.

The input 2D data residing on global memory is divided into many contiguous blocks as Fig 4.5 shows, with overlapping areas to account for the halo layers. Since the cache size is considerably smaller than the original 2D data, the halo data must be stored into cache memory multiple times. Note that the required halo data is loaded from global memory and stored to cache and incurs no additional computation: the halo data does not generate redundancy in the convolution computation.

The following parameters are defined in our analysis: T_{g2rc} is the required time of loading data from global memory to register in the case of using the register cache method. When using the shared memory method, T_{g2rs} is the required time for loading the data from global memory to registers and T_{r2s} is required time of storing data from register to shared memory.

Loading the halo layer data is a redundant operation that could penalize performance. We compare the penalizing effects of halo layers between SSAM and shared memory. We define the ratio of halo for SSAM as $HR_{rc} = (S{\cdot}C-(S-M){\cdot}(C-N))/(S{\cdot}C)$, where S is WarpSize. Hence, we can further derive that $HR_{rc} < (S{\cdot}N+C{\cdot}M)/(S{\cdot}C)$. We define HR_{smc} as the ratio of shared memory used for the halo layers, where $HR_{smc}\in[0,1)$, we can reach that $T_{g2rc}/(1+HR_{rc})$ approximates $T_{g2rs}/(1+HR_{smc})$. Since the shared memory can be accessed within a CUDA block, while the register can be only within a warp, $HR_{rc}\gg HR_{smc}$ becomes true. Suppose that the shared memory caches the global memory without halo layers as HR_{smc}=0, we can achieve the ideal caching pattern as $T_{g2rc}\approx(1+HR_{rc}){\cdot}T_{g2rs}$. The difference in time between using the shared memory method versus the register cache for loading and storing data may be expressed as $T_{dif\_mem\_io} = (T_{g2rs}+T_{r2s}+T_{r2g})-(T_{g2rc}+T_{r2g})$, where T_{g2rc} is the time required to store the convolution result of W*H elements back to the global memory. Hence, $T_{dif\_mem\_io} = T_{r2s}-T_{g2rs}{\cdot}HR_{rc}$. Suppose that each thread processes P elements and loads C elements, the difference of using shared memory and register cache to compute convolution, while accounting for the halo layers is

$$\begin{aligned} Dif &\approx T_{r2s}-T_{g2rs}{\cdot}HR_{rc}+P{\cdot}Dif\_smem\_reg \\ &> T_{r2s}-T_{g2rs}{\cdot}(N/(N+P-1)+M/32)+P{\cdot}(M{\cdot}N{\cdot}T_{smem\_read}-(M-1){\cdot}T_{shfl}) \end{aligned}$$

for each thread loading C elements from global memory. The difference of execution time on average can be expressed as

$$\begin{aligned} AvgDif = Dif/C > T_{r2s}/C-T_{g2rs}/C{\cdot}1/(N/(N+P-1)+M/32) \\ +P/C{\cdot}(M{\cdot}N{\cdot}T_{smem\_read}-(M-1){\cdot}T_{shfl}) \end{aligned}$$

In comparison to HR_{rc}, HR_{smc} is relatively small since T_{r2s}/C is purely dominated by the shared memory latency. Considering T_{g2rs}/C as the global memory read latency, we reach

$(T_{r2s}/C) = T_{smem} \gg T_{smem\_read}$ and $(T_{g2rs}/C) \approx T_{gmem\_read}$, thus

$$AvgDif > T_{smem\_read} - T_{gmem\_read} \cdot (N/(N+P-1) + M/32) + P \cdot M \cdot N \cdot T_{smem\_read}/(N+P-1) - (M-1) \cdot T_{shfl}$$

Since T_{gmem_read} is 200$\sim$400 cycles/warp for coalesced access [118]. We can conclude that $AvgDif \gg 0$, where M$\geq$2 and N$\geq$2.

To sum up, in comparison to the shared memory-based algorithms, the overhead of handling the halo layers in register cache method is marginal.

4.4.4 The Importance of Dependency *D* in SSAM

In this section, we present the importance of dependency (namely D) in SSAM. Our performance model proves the bases for why the SSAM improves the performance of memory-bound kernel. Based on former discussions and the Equation 3.2, we can conclude the following:

(i) The in-register partial sums computing operation (O) achieves higher computing efficiency in comparison to the conventional methods.

(ii) Coalescing access to global memory and efficient register cache make the input (X) and the output (Y) operations perform data Read/Write efficiently.

(iii) However, the partial sums transfer path (D) varies from one algorithm to another. Exploring the correct dependency is critical to algorithm performance and thus we need a careful design of the data transfer paths in SSAM. In other words, D should be mapped carefully to the register communication pattern within a single warp, e.g. Fig 3.1 shows that exchanging registers in the horizontal direction is more expensive than vertical. Hence, decreasing the transfer of partial sums in the horizontal direction is essential for expressing algorithms efficiently in SSAM. More specifically, we can determine the best D by computing and comparing the latency of variants of SSAM-based kernels via micro-benchmarking, e.g. computing the latency of 2D convolution as in Equation 4.3.

4.5 Evaluation

This section is dedicated to reporting the achieved performance of our execution model using the latest Nvidia GPUs. The performance of 2D convolution is analyzed in detail.

4.5.1 Software & Hardware Setup

The experimental results presented here are evaluated by two Tesla P100 and V100 GPUs. The CUDA driver is 410.48, we use CUDA 10.0 (including NPP/cuFFT and cuDNN v7.4), GPU memory is 16GB, and the OS is CentOS 7.6. ECC option for GPUs is on. Nvidia nvcc and gcc-5.4 are used to compile the CUDA kernel and host codes, respectively. The applied compiler option of nvcc for Pascal and Volta architectures is as follows : -gencode arch=compute_60,code=sm_60 -gencode arch=compute_70,code=sm_70.

4.5.2 2D Convolution Results

In Figure 4.6, we evaluate 2D convolutions using single-precision for various filter sizes, which ranges from 2×2 to 20×20. The input image size is fixed as 8192×8192. It is noteworthy that P=4, B=128 are used as parameters. We use the OS-independent *cudaEvent* function to measure the execution time for all of the convolution computations, and ignore the data transfer time between host and device. Note that although in our experiments we only show square-shaped filters (i.e. M=N), a simple change in a template function in our implementation enables the computation of 2D convolution for any filter shape (M≠N) as well.

(i) **ArrayFire** [168]. ArrayFire is a highly optimized library by CUDA. We report the performance of the fastest 2D convolution kernel, called *kernel::convolve2*, in which the shared memory and constant memory are employed to cache image data and filter weights, respectively. Its filter size limitation is 16×16. Note that there are no official documents about this value, which was found out by analyzing source code and lots of tests.

(ii) **NPP** [30]. The Nvidia Performance Primitives (NPP) library is a closed-source library, thus we investigate it by the *nvprof* profiling tool. NPP does not use any shared memory as cache. Note that NPP particularly optimizes the convolution computation for filters of size 3×3 and 5×5 using dedicated kernels, the kernel names are *FilterBorder32f3x3ReplicateQuadNew* and *FilterBorder32f5x5ReplicateQuadNew*, respectively.

(iii) **cuFFT** [30]. The Nvidia CUDA Fast Fourier Transform (cuFFT) library provides a constant, yet relatively high, run time regardless of the filter size.

(iv) **Halide** [111]. The Halide is a domain-specific language (DSL) designed to generates pipelines for image processing [111]. It is difficult to instrument its pipeline to measure

the kernel execution time, so we use the Nvidia *nvprof* profiling tool to report the kernel execution time.

(v) **cuDNN** [24]. cuDNN is a GPU-accelerated library by Nvidia for deep neural networks. We evaluate it by a special parameter that an image with a single channel is convolved by a single filter. Several algorithms are implemented in cuDNN, and we only report the result with the best performance. Note that the supported filter sizes in cuDNN must be odd numbers, e.g. 3×3, 5×5.

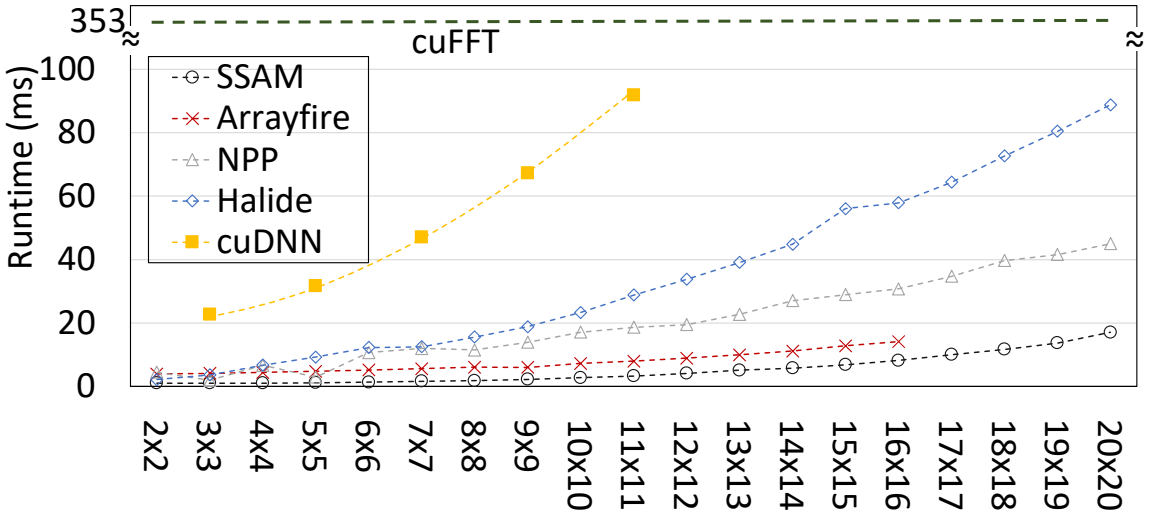
Figure 4.6 gives indications of the accuracy of the findings of the performance model (Section 4.4). Equation 4.4 indicates that given a fixed output size of 2D convolution, while the increasing size of filters, the performance difference between SSAM and other kinds of implementations (not only shared memory-based algorithm) becomes increasingly larger. Such a conclusion can be verified by observing the difference in the performance of the graphs in Figure 4.6.

4.6 Summary

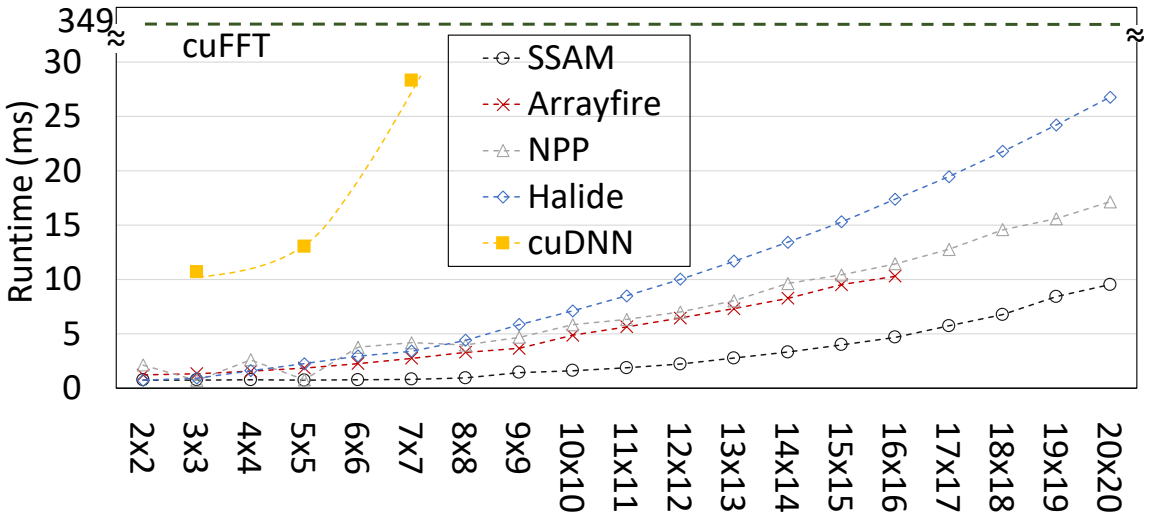
We present the implementation of the SSAM-based 2D convolution algorithm. All key techniques of the SSAM are used in our implementation. We use registers and shared memory to cache the image data and filter weights, respectively. The register cache improves the data locality and increases the data reuse for the in-register computation. We adopt overlapped tiling to increase the parallelism and decrease the intra-warp divergency. The partial sums are accumulated efficiently due to the in-register computation. Furthermore, the partial sums are shifted within the virtual systolic array efficiently by the shuffle intrinsic.

To justify the validation of the achieved performance, we build a performance model and measure some of the constant variables in GPUs, e.g. the latency of shuffle intrinsic. Meanwhile, we prove the overhead of the halo-layer is insignificant in our implementation.

Using the latest Tesla P100 and V100 GPUs, we compare the SSAM with a lot of state-of-the-art implementations, e.g. Nvidia NPP, ArrayFire. Our implementation is as simple as a single kernel, yet outperforms other libraries. Our implementation can be used in a large variant of image processing applications to meet the requirement of real-time processing.



(a) Single precision evaluation on P100. cuFFT method is constant as 353 ms., the x-axis is filter size, the y-axis is execution time.



(b) Single precision evaluation on V100. cuFFT method is constant as 349 ms. The image size is 8192×8192, the x-axis is filter size, the y-axis is execution time.

Fig. 4.6 2D Convolution performance and scalability.

Chapter 5

Stencils in SSAM

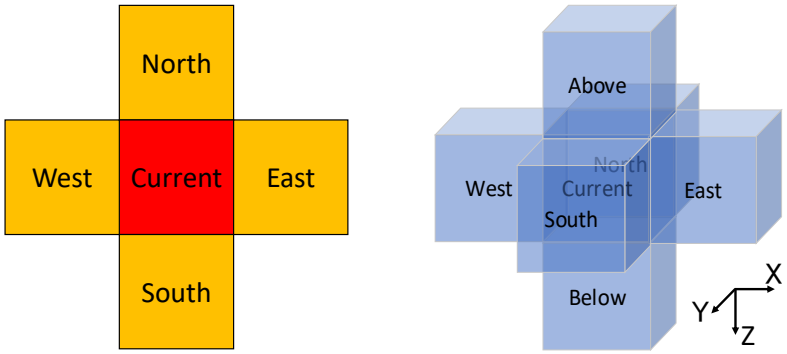
In this chapter, we firstly introduce stencil computation and discusses its importance, then review the related work and present how to map the stencils in SSAM. Finally, we evaluate the performance of the SSAM-based 2D/3D stencil implementations using spatial blocking and temporal blocking techniques.

5.1 Background & Motivation

Stencil computations are one of the core problems of scientific applications in many domains [108, 71] and their performance is critical to the scientific community. Several works are targeting optimizing stencils using the latest accelerators, i.e. many-core CPUs, Intel Xeon-Phi, FPGA, GPUs. In this study, we focus on Nvidia GPUs due to GPUs is widely applied accelerator in many fields, e.g. HPC, AI, etc. A lot of effective methodologies are achieved to speed up the stencil computation, e.g. spatial blocking (as in Section 5.1.3), temporal blocking (as in 5.1.4). Such kinds of algorithms are mostly employed to improve data locality and reduce access to global memory. Spatial blocking, also known as tiling, is a significant transformation to enhance the temporal reuse of data and thus, can be used for the reduction of the amount of data movement between host and GPUs. Regarding the memory-bound kernels, this kind of approach is a straightforward way to improve the performance of stencil kernels. Many recent efforts focus on GPU register optization [132, 134], i.e. tuning register allocation, alleviating register pressure, etc.

5.1.1 2D Stencils

In 2D stencils, each cell is typically updated by the inner product of a set of stencil coefficients and its neighboring cells' values, which are determined by the stencil shape and radius.



(a) 2D 5-point stencil. (b) 3D 7-point stencil.

Fig. 5.1 2D/3D stencils.

Figure 5.1a shows a typical first-order 2D diffusion 5-point stencil (also known as 2D Jacobi stencil). In each iteration, the dual buffers are applied by swapping pointers as to the input/output or output/input buffers. Mathematically, the 2D diffusion stencil is defined as

$$\begin{aligned} s_{i+1}(x,y) = {} & s_i(x-1,y)\cdot West + s_i(x,y-1)\cdot North \\ & + s_i(x,y)\cdot Current + s_i(x,y+1)\cdot South + s_i(x+1,y)\cdot East \end{aligned}$$

where (x, y) is a relative position, s_i is a cell's value at the i^{th} iteration, and $\cdot$ is the multiplication operator. West, North, Current, South, and East denote the stencil coefficients.

5.1.2 3D Stencils

Regarding 3D stencils, taking the 7-point diffusion stencil as an example (Figure 5.1b), two vertical points (namely Above and Below) are added to the 2D 5-point stencil. The stencil is computed by adding $s_i(x,y,z-1)\cdot Above$ and $s_i(x,y,z-1)\cdot Below$, where (x, y, z) is the cell position. It is worth mentioning that different stencil shapes/sizes appear in scientific simulations.

The 3D stencil can be extended from the previous 2D stencil. Same to 2D stencil, 3D-7pt stencil can be computed iteratively as:

$$\begin{aligned} s_{i+1}(x,y,z) = {} & s_i(x-1,y,z) * West + s_i(x,y-1,z) * North \\ & + s_i(x,y,z) * Current + s_i(x,y+1,z) * South \\ & + s_i(x+1,y,z) * East + s_i(x,y,z-1) * Above \\ & + s_i(x,y,z-1) * Below \end{aligned} \tag{5.1}$$

where (x, y, z) means position, s_i is a cell's value of at the i^{th} iteration, $West$, $North$, $Current$, $South$, $East$, $Above$ and $Below$ are seven points of stencil coefficients.

5.1.3 Spatial Blocking Methodology

For typical stencil computations, due to the neighboring cells are regularly reused one or more times, the grids are carefully blocked in multiple dimensions, which result in full spatial reuse and significant reduction of redundant memory access to the high-latency memory, e.g. global memory in CUDA. In this scenario, the redundant accesses will occur at the boundaries of the blocks. This approach is termed as spatial blocking (or tiling). Such a blocking technique is an effective way to improve the data locality due to the neighboring cells are reused in stencil computations. Furthermore, blocking can significantly improve the concurrency in time-tiled stencils as illustrated in [174, 74].

5.1.4 Temporal Blocking Methodology

The temporal block method was proposed by Leonardo et al. [102, 103] to overcome the GPU memory limitation while overlapping sub-domains (like our overlapped blocking in Section 4.3.5). More specifically, to reduce the cost of data movement between host and GPUs, the domain is decomposed to several small sub-domains, which are moved to GPUs for computing multiple time steps. Even using the spatial blocking technique, most of the stencil computations are still bounded by the memory traffic in the latest accelerators. Such a scenario motivates researchers to further optimize the stencil computations by reducing the required memory bandwidth. There are two characteristics in the real stencils. One is that the computation relies on thousands of iterative computations; the other is the read-after-write dependency between the iterations. Hence, it makes sense to extend the temporal blocking on the top of spatial blocking to further reduce the required external memory accesses in stencil computations.

5.2 Implementation of stencil in SSAM

In this section, we focus on the implementation of the stencil in SSAM. Similar to mapping the 2D convolution algorithm in SSAM, we implement efficient yet simple stencil kernels using the key techniques of SSAM, e.g. register cache, partial sums, shuffle intrinsic, etc. We will analyze in the later sections, the optimization must meet the requirement of CUDA architecture as follows : Access global memory in a coalesced fashion; To tolerate the latency of memory access, launch sufficient concurrency; Increase the data reuse in the fast on-chip

memory, e.g. scratchpad, registers; Perform data communication between threads efficiently. we present the details of SSAM-based 2D/3D stencils.

5.2.1 Mapping 2D Stencil to SSAM

In this section, we describe mapping 2D stencils to SSAM. In Listing 5.1, all variable definitions are the same as Listing 5.1, i.e. T, B, P, W, H, C, etc... *West*, *North*, *Current*, *South* and *East* are stencil coefficients. The definitions of other variables are same to Listing 4.1. In Listing 5.1, the SSAM-based 5-point stencil kernel differs slightly from the convolution

Listing 5.1 2D 5-point stencil CUDA kernel.

```
template<typename T, int B, int P>
__global__ void 2d5pt(const T* src, T* dst, int W, int H)){
   const int C = P + N - 1;
    //register files
   T data[C];
   //1, Load data from global memory to registers
    data[...] = src[...];
   //2, Compute Stencil Results
   #pragma unroll
   for (int i=0; i<P; i++) {
      T sum = 0;
        //1st column
        sum = MAD(data[i + 1], West, sum);
        sum = __shfl_up_sync(0xffffffff, sum, 1);
        //2nd column
        sum = MAD(data[i + 0], North, sum);
        sum = MAD(data[i + 1], Current, sum);
        sum = MAD(data[i + 2], South, sum);
        sum = __shfl_up_sync(0xffffffff, sum, 1);
        //3th column
        sum = MAD(data[i + 1], East, sum);
        data[i] = sum;
   }
    //3, Store Result to Global Memory
    dst[...] = data[...];
}
```

example in Listing 4.1. First, the stencil coefficients are divided into three groups as {*West*}, {*North*, *Current*, *South*}, and {*East*}. Then we compute in parallel the partial sums between the coefficients groups and the cached data. Finally, the partial sums are shifted to the neighbor threads like Section 4.3.3. In 2D convolution, we cache the filter coefficients by shared memory to account for the cases of large filters. Stencils typically have fewer coefficients than convolutions, so we directly transfer the stencil coefficients to the kernel as

arguments (namely *Current*, ..., *East*). It is important to note that SSAM is not limited to low order stencils such as the 5-point stencil. SSAM, as will be shown, is highly effective for different shapes of stencils of the high order.

This paragraph describes the detailed implementation of 2d5pt stencil implementation in Listing 5.1 as follows:

(i) A subset of the cells' data residing in global memory is cached into registers (lines 6∼7).

(ii) According to the sliding window position, we compute the partial sums by CUDA MAD intrinsic and transfer them between threads by shuffle intrinsic (lines 13∼21). Note that sliding window scheme is same to 2D convolution as in Listing 4.1

(iii) We move the sliding window step by step for a total of P times (line 10).

(iv) Finally, the computed results, which reside in the register cache, are stored back to global memory (lines 24∼25).

It is worth mentioning that we ignore the details of the uppon steps on computing and shift partial sum due to they are same as the implementation of 2D convolution as illustrated in Section 4.3.1.

5.2.2 Mapping 2D Stencil to SSAM by temporal blocking

This section describes how to map 2D stencil to SSAM by temporal blocking technique. In Listing 5.2, we show the SSAM-based 2D stencil implementation by temporal blocking technique (called 2d5pt_tb). the definition of the parameters is same to the earlier illustration in Listing 4.1, i.e. T, B, P, W, H, C, etc.. The S is defined as the temporal time-steps and thus the line 9 of Listing 5.2 indicates the iteration of 2d5pt stencil computation repeats S times. Note that the 2d5pt stencil computation for a single iteration (line 11∼24) is full same to Listing 5.1.

5.2.3 Mapping 3D Stencil to SSAM

This section discusses how to map 3D Stencil to SSAM. We divide the 3D grid into many 3D sub-grids with overlapping blocks (to account for the halo layers as in Figure 4.5). Each sub-grid is processed by a CUDA block, and each warp in a CUDA block processes a 2D slice in the X-Y plane as in Figure 5.1b. Regarding the threads accumulating the final result $s_{i+1}(x,y,z)$, since in the Z direction values of $s_i(x,y,z-1)$ and $s_i(x,y,z+1)$ are inaccessible

Listing 5.2 2D 5-point stencil CUDA kernel by temporal blocking.

```
template<typename T, int B, int P, int S>
__global__ void 2d5pt_tb(const T* src, T* dst, int W, int H)){
    const int C = P + N - 1;
     //register files
    T data[C];
   //1, Load data from global memory to registers
    data[...] = src[...];
    #pragma unroll
    for (int s=0; s<S; s++){ //S is iterative time-steps
      #pragma unroll
      for (int i=0; i<P-s*order*2; i++) { //order is stencil order
            T sum = 0;
            //#1st column
            sum = MAD(data[i + 1], West, sum);
            sum = __shfl_up_sync(0xffffffff, sum, 1);
            //#2nd column
            sum = MAD(data[i + 0], North, sum);
            sum = MAD(data[i + 1], Current, sum);
            sum = MAD(data[i + 2], South, sum);
            sum = __shfl_up_sync(0xffffffff, sum, 1);
            //#3rd column
            sum = MAD(data[i + 1], East, sum);
            data[i] = sum;
      }
    }
    //3, Store Result to Global Memory
    dst[...] = data[...];
}
```

(i.e. residing the registers of neighbor warps), we use the shared memory to accumulate the partial sums, which are computed by corresponding warps. In this scenario, SSAM performs intra-warp communication using the shuffle instruction and inter-warp communication using shared memory.

In Listing 5.3, Listing 5.4, Listing 5.5, and Listing 5.6, the *src* and *dst* are defined as input and output of stencils, respectively. The *data* and *sMem* are registers and shared memory, respectively. *_Fxxx* is stencil coeficients, e.g. *_F111* is the *Center* as introduced in Section 5.1.2, *_F110* is *West*, *_F112* is *East*, *_F110* is *North*, *_F112* is *South*, *_F011* is *Above*, *_F211* is *Bottom*. It is noteworthy that we mixed *__shuffle_up* and *__shuffle_down* in all implementations to emphas that any shuffle intrinsic can be used in SSAM for transferring the partial sums within a single warp. The difference between *__shuffle_up*, *__shuffle_down*, and *__shuffle* can be found in Section 2.1.4.

Listing 5.3 3d7pt stencil CUDA kernel.

```
//1, load data from global memory to register and shared memroy
data[...] = src[...][...][...];
sMem[...][...][...] = data[...];
__syncthreads()

for (int i = 0; i < P; i++) {
    T sum = 0;
    //1st column
    sum = MAD(data[i + 1], _F110, sum);
    sum = __shfl_up_sync(0xffffffff, sum, 2);
    //3th column
    sum = MAD(data[i + 1], _F112, sum);
    sum = __shfl_down_sync(0xffffffff, sum, 1);
    //2nd column
    sum = MAD(data[i + 0], _F101, sum);
    sum = MAD(data[i + 1], _F111, sum);
    sum = MAD(data[i + 2], _F121, sum);
    //top & bottom
    sum += sMem[...][...][...] * _F011;
    sum += sMem[...][...][...] * _F211;
    data[i] = sum;
}
//store to global memory
dst[...][...][...]=data[...];
```

In Listing 5.3 and Listing 5.4, we present the SSAM-based 3d7pt stencil computation using spatial blocking and temporal blocking, respectively. In Listing 5.5 and Listing 5.6, we list the SSAM-based 3d19pt stencil computation using spatial blocking and temporal blocking, respectively. Similar to the 2D convolution in Section 4.3.1 and 2D stencils in the earlier sections, our 3D stencil implementations perform in the same workflow that loading data from global memory to registers, performing the SSAM computation as illustrated in Section 4.3.1, and storing data from registers to global memory. Hence, it makes sense that the SSAM code can be automated by DSL or compiler despite we do not show the real implementation in this study.

5.3 Evaluation

This section is dedicated to reporting the achieved performance of SSAM-optimized stencils using the latest Nvidia GPUs. The performance of 2D/3D stencil computation with Temporal/Spatial Blocking technique is analyzed in detail.

Listing 5.4 3d7pt stencil CUDA kernel by temporal blocking.

```
//1, load data from global memory to register and shared memroy
data[...]=src[...][...][...];
for (int s = 0; s < S; s++) {
   //load data to shared memory
   sMem[...][...][...] = data[...];
   __syncthreads()

    for (int i = 0; i < P-order*2*s; i++) {
        T sum = 0;
        //1st column
        sum = MAD(data[i + 1], _F110, sum);
        sum = __shfl_up_sync(0xffffffff, sum, 2);
        //3rd column
        sum = MAD(data[i + 1], _F112, sum);
        sum = __shfl_down_sync(0xffffffff, sum, 1);
        //2nd column
        sum = MAD(data[i + 0], _F101, sum);
        sum = MAD(data[i + 1], _F111, sum);
        sum = MAD(data[i + 2], _F121, sum);
        //top & bottom
        sum += sMem[...][...][...] * _F011;
        sum += sMem[...][...][...] * _F211;
        data[i] = sum;
    }
    __syncthreads(); //sync threads
}
//store to global memory
dst[...][...][...]=data[...];
```

5.3.1 Stencil Results

Table 5.1 Stencil benchmark.

benchmark	k	FPP	benchmark	k	FPP	benchmark	k	FPP
2d5pt	1	9	2d9pt	2	17	2d13pt	3	25
2d17pt	4	33	2d21pt	5	41	2ds25pt	6	49
2d25pt	2	33	2d64pt	4	73	2d81pt	4	95
2d121pt	5	241	3d7pt	1	13	3d13pt	2	25
3d27pt	1	30	3d125pt	2	130	poisson	1	21

In Table 5.1, the FPP is FLOP per point. The domain sizes of the 2D and 3D stencil are 8192^2 and 512^3, respectively. A detailed description of the benchmarks can be found in literatures [132, 133].

In Figure 5.2, the x-axis is the stencil benchmarks defined in Table 5.1. The y-axis is the performance in a unit of GCells/s. The "original", "reordered", and "unrolled" are implemented by Rawat et al. as open-source [132, 133], the "ppcg" is auto-generated from the C code using the ppcg compiler [151].

To better understand the performance of SSAM-based stencil computations, we evaluate a diverse collection of 2D/3D stencil benchmark (listed in Table 5.1) and compare the performance with a variety of CUDA implementations on the latest GPUs. Note that the stencils in the experiments include both low order and high order stencils. High-performance stencil libraries rarely provide a consistent performance advantage for both low and high

Listing 5.5 3D 19-point stencil CUDA kernel.

```
//load data from global memory to registers
data[...] = src[...][...][...];
for (int i=0; i<P; i++){  //1st SSAM
    T sum = 0;
    sum = MAD(data[i + 1], _F110, sum); //1st column
    sum = __shfl_up_sync(0xffffffff, sum, 2);
    sum = MAD(data[i + 1], _F112, sum);//3th column
    sum = __shfl_down_sync(0xffffffff, um, 1);
    sum = MAD(data[i + 0], _F101, sum);//2nd column
    sum = MAD(data[i + 1], _F111, sum);
    sum = MAD(data[i + 2], _F121, sum);
    sMem[...][...][...] = sum;         //accumulate
}
__syncthreads();                       //sync threads
for (int i=0; i<P; i++){ //3th SSAM
   T sum = 0;
   sum = MAD(data[i + 1], _F210, sum);//1st column
   sum = __shfl_up_sync(0xffffffff, sum, 2);
   sum = MAD(data[i + 1], _F212, sum);//3th column
   sum = __shfl_down_sync(0xffffffff, um, 1);
   sum = MAD(data[i + 0], _F201, sum);//2nd column
   sum = MAD(data[i + 1], _F211, sum);
   sum = MAD(data[i + 2], _F221, sum);
   sMem[...][...][...] += sum;       //accumulate
}
__syncthreads();                      //sync threads
for (int i=0; i<P; i++){ //2nd SSAM
   T sum = 0;
   sum = MAD(data[i + 0], _F100, sum);//1st column
   sum = MAD(data[i + 1], _F110, sum);
   sum = MAD(data[i + 2], _F120, sum);
   sum = __shfl_up_sync(0xffffffff, sum, 2);
   sum = MAD(data[i + 0], _F102, sum);//3th column
   sum = MAD(data[i + 1], _F112, sum);
   sum = MAD(data[i + 2], _F122, sum);
   sum = __shfl_down_sync(0xffffffff, um, 1);
   sum = MAD(data[i + 0], _F101, sum);//2nd column
   sum = MAD(data[i + 1], _F111, sum);
   sum = MAD(data[i + 2], _F121, sum);
   data[i] = sMem[...][...][...] + sum;//accumulate
}
//store data from registers to global memory
dst[...][...][...]=data[...];
```

order stencils since high order stencils are typically bound by the registers. For instance, temporal blocking is effective for low order stencils [174], while register re-ordering schemes are effective for high order stencils [132, 133].

Figure 5.2 shows three implementations: "*original*", "*reordered*", and "*unrolled*", with state-of-the-art performance results by Rawat et al. [132, 133]. The "*original*" is a basic CUDA implementation," *reordered*" is register-optimized version for "*original*", "*unroll*" is the unroll-optimized method. Note that we adapted Rawat's source code to P100/V100 GPUs, ran multiple tuned configurations and only report the result with the best performance. We also compare with the latest *ppcg* (0.08 version) compiler [151] and Halide [111].

We use GCells/s and not GFlops/s as the performance metric in all our stencil experiments since we observed that different libraries count FLOPS differently. Additionally, the achieved GFlops/s can be obtained by multiplying the reported GCells/s with the corresponding FPP factor (as in Table 5.1). Note that the execution time of all stencil kernels is measured by Nvidia *nvprof* profiler.

Listing 5.6 3D 19-point stencil CUDA kernel by temporal blocking.

```
//load data from global memory to registers
data[...] = src[...][...][...];
for (int s = 0; s < S; s++) {
    sMem[...][...][...] = data[...];
    for (int i=0; i<P-2*order*s; i++){  //1st SSAM
        T sum = 0;
        sum = MAD(data[i + 1], _F110, sum); //1st column
        sum = __shfl_up_sync(0xffffffff, sum, 2);
        sum = MAD(data[i + 1], _F112, sum);//3th column
        sum = __shfl_down_sync(0xffffffff, sum, 1);
        sum = MAD(data[i + 0], _F101, sum);//2nd column
        sum = MAD(data[i + 1], _F111, sum);
        sum = MAD(data[i + 2], _F121, sum);
        sMem[...][...][...] = sum;         //accumulate
    }
    __syncthreads();                       //sync threads
    for (int i=0; i<P-2*order*s; i++){ //3th SSAM
      T sum = 0;
      sum = MAD(data[i + 1], _F210, sum);//1st column
      sum = __shfl_up_sync(0xffffffff, sum, 2);
      sum = MAD(data[i + 1], _F212, sum);//3th column
      sum = __shfl_down_sync(0xffffffff, sum, 1);
      sum = MAD(data[i + 0], _F201, sum);//2nd column
      sum = MAD(data[i + 1], _F211, sum);
      sum = MAD(data[i + 2], _F221, sum);
      sMem[...][...][...] += sum;       //accumulate
    }
    __syncthreads();                       //sync threads
    for (int i=0; i<P-2*order*s; i++){ //2nd SSAM
      T sum = 0;
      sum = MAD(data[i + 0], _F100, sum);//1st column
      sum = MAD(data[i + 1], _F110, sum);
      sum = MAD(data[i + 2], _F120, sum);
      sum = __shfl_up_sync(0xffffffff, sum, 2);
      sum = MAD(data[i + 0], _F102, sum);//3th column
      sum = MAD(data[i + 1], _F112, sum);
      sum = MAD(data[i + 2], _F122, sum);
      sum = __shfl_down_sync(0xffffffff, sum, 1);
      sum = MAD(data[i + 0], _F101, sum);//2nd column
      sum = MAD(data[i + 1], _F111, sum);
      sum = MAD(data[i + 2], _F121, sum);
       data[i] = sMem[...][...][...] + sum;//accumulate
    }
    __syncthreads();
}
//store data from registers to global memory
dst[...][...][...]=data[...];
```

In Figure 5.2, SSAM mostly outperforms the other implementations using the latest GPUs in both single and double precision runs. The performance advantage of SSAM is due to the better register locality, less global memory access and efficient threads communication. Such a result is consistent with the performance of 2D convolution (as in Figure 4.6) and is further explained by the performance model.

There are two important observations in Figure 5.2. One is that in comparison to the P100 GPU, the performance variance on V100 become smaller, and the other is that when using double precision on the V100 GPU, few of the SSAM-based stencils do not achieve the highest performance. The reason behind that is that Volta's architecture is different in many aspects, which will be discussed in Sec 8.1.3.

5.3.2 Comparison with Temporal/Spatial Blocking Libraries

SSAM is a versatile model that enables temporal blocking without much change to the implementation: the use of register cache and shuffling in SSAM does not limit the use of temporal blocking. It is worth to mention that temporal/spatial blocking algorithms are widely used to improve the performance of stencil computation by reducing the required memory bandwidth. In this section, we focus on comparing SSAM to state-of-the-art temporal blocking libraries for stencils.

StencilGen We compare the performance with **StencilGen** [134]. To the author's knowledge, StencilGen reports the highest performance among temporal blocking libraries. As Figure 5.3 shows, in the majority of benchmarks, SSAM performs better than StencilGen. However, register pressure can limit our performance in some cases. It is important to mention that StencilGen is optimized for stencil computations only, while SSAM is more versatile and general for different types of kernels. Besides, it is important to point out that SSAM provides a consistent performance advantage for both low and high order stencils. Some approaches, such as those proposed in [133, 134], can further be adopted by SSAM to improve the performance for the specific cases of deep temporal blocking.

Diffusion The authors in [174] report a competitive performance of the 3d7pt stencil (highly optimized by shared memory as proposed in [101], called **Diffusion** in Figure 5.3) as 92.7 GCells/s and 162.4 GCells/s using single precision on P100 and V100 GPUs, 30.6 GCells/s and 46.9 GCells/s using double precision on P100 and V100 GPUs, respectively. This performance is significantly lower than SSAM.

Bricks **Bricks** is a general stencil library targeting both CPUs and GPUs [173]. The performance of Bricks on P100 is reported to be 41.4 GCells/s and 24.25 GCells/s for single and double precision, respectively. As seen in Figure 5.3a and Figure 5.3b, SSAM outperforms Bricks on P100. Since Bricks is not publicly available, we can not compare our result for the V100 GPU.

5.4 Summary

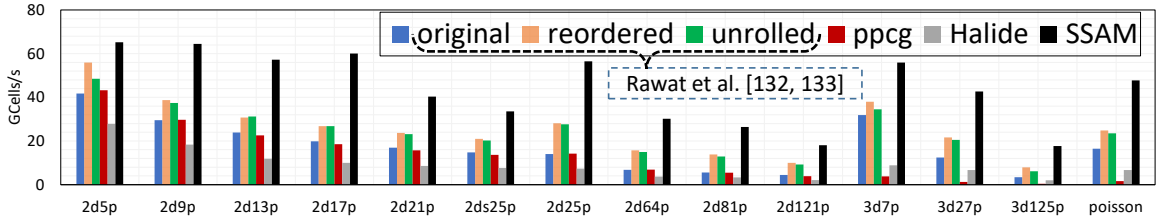
In this chapter, we introduce the importance of stencil computation and map a variant of stencil kernels in SSAM using spatial blocking and temporal blocking techniques. We evaluate our implementations on the latest Nvidia Tesla GPUs and compare the performance with the state-of-the-art libraries.

5.4.1 Limitation

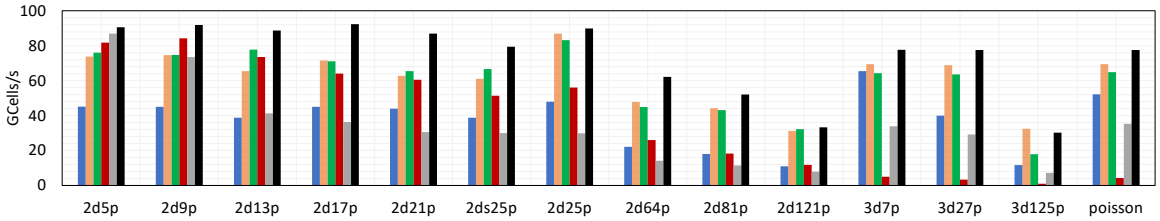
Though we achieve outstanding performance for stencil computations in some of the computational patterns as the earlier section shows, there are some limitations in SSAM-based stencils that should be taken into considerations. Since we fully rely on registers to cache data and perform the in-register computation, register pressure limited the performance scaling when using more registers. To be more precise, the registers we used are register variables in all implementations and is not the physical GPU registers, because the NVCC compiler is closed-source, the register allocation is black-box to programmers. Note that PTX, a parallel thread execution (PTX) instruction set architecture (ISA) for programming CUDA applications, performs as an intermediate representation as CUDA C/C++ without control of the physical registers on GPUs. It results that the versions of both CUDA driver and NVCC compiler may affect the final performance of our implementation to some extent.

5.4.2 On-going work

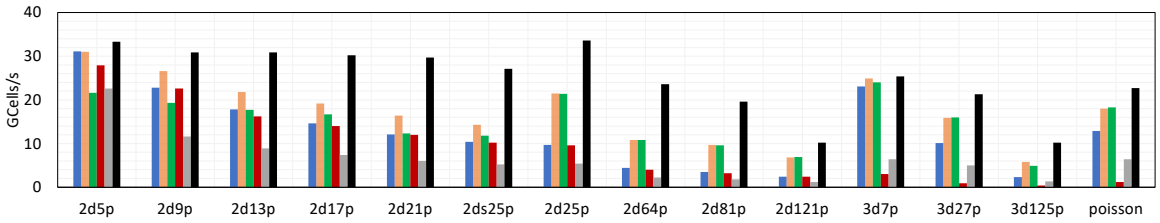
Using stencil as an illustrative example, we justify the versatility of SSAM model. Though there are many limitations in SSAM as analyzed in the earlier section, the stencil kernels that are mapped in SSAM are very simple yet highly efficient in computation. Meanwhile, there are several directions in future work as follows. (i) Due to the generality of SSAM, we can automate the SSAM-based code by polyhedral analysis as will be discussed in Section 8.1.1. (ii) We only demonstrate the stencil kernel with simple input that the cells are organized a single array, it is required to tackle complex stencils as presented in literature [132, 133], e.g. derivative [107], SW4 (Geodynamics Seismic Wave SW4 application) [38]. (iii) We can integrate the SSAM-based kernels into real-world stencil applications. (iv) Optimizing the usage of the on-chip resource (e.g. registers, scratchpad memory) is crucial to the overall performance of the GPUs, finding out the balance between such resources and performance is a promising and significant work. (v) We plan to leverage SSAM to FPGA applications by high-level synthesis as in literature [35], e.g. OpenCL.



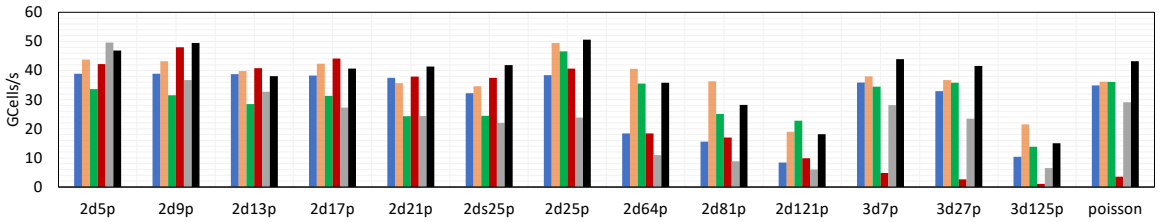
(a) Single precision on Tesla P100 GPU.



(b) Single precision on Tesla V100 GPU.



(c) Double precision on Tesla P100 GPU.



(d) Double precision on Tesla V100 GPU.

Fig. 5.2 Performance evaluation for the 2D and 3D stencil benchmarks on Tesla P100 and V100 GPUs. the x-axis is the stencil benchmarks defined in Table 5.1. The y-axis is the performance in a unit of GCells/s.

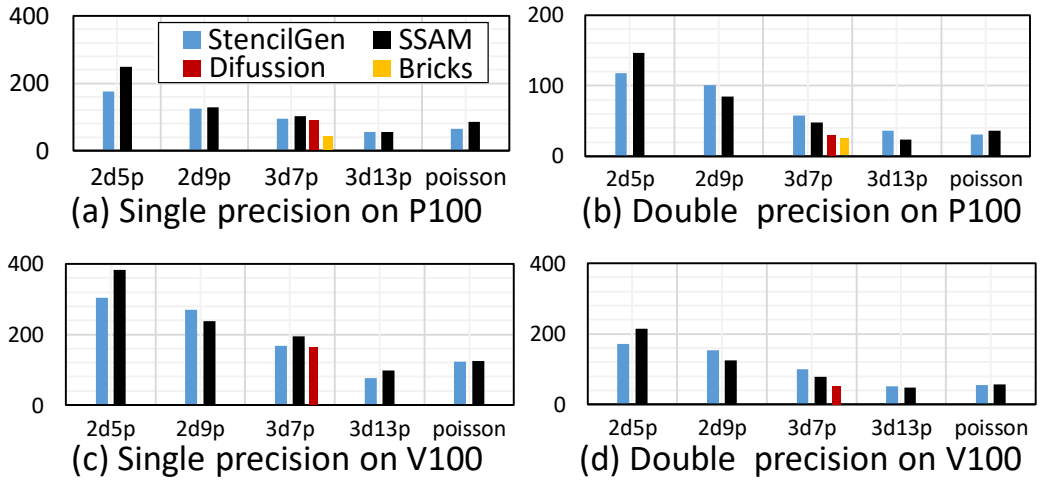


Fig. 5.3 Performance evaluation for the 2D and 3D stencil computation using temporal blocking. the x-axis is stencil benchmark, the y-axis is the performance in the unit of GCells/s.

Chapter 6

Summed Area Tables Primitive in SSAM

In this chapter, we first provide the background of Summed Area Tables primitive, e.g. the definition of SAT, the SAT-based applications. Secondly, we introduce the one-dimensional scan operator (e.g. parallel scan and serial scan), and the two-dimensional scan (or SAT). Thirdly, we present the proposed algorithm by illustrating the details of mapping SAT computation in SSAM, i.e. ScanRow-BRLT, BRLT-ScanRow, and ScanRowColumn. Fourthly, we employ a performance model to prove the advance of the BRLT algorithm, which transposes a batch of matrixes of size 32×32. Fifthly, using the latest Nvidia GPUs (P100 and V100), we demonstrate the computational efficiency of the proposed algorithm by comparing the performance with two state-of-the-art implementations, namely OpenCV and NPP. Finally, we conclude our work and show the direction of future work for SAT-related applications.

6.1 Background

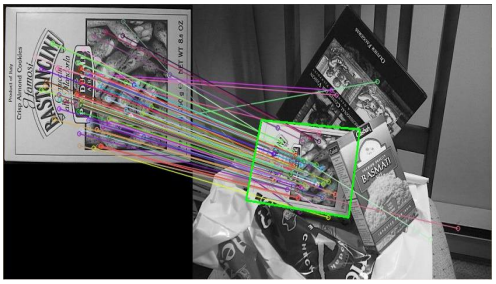
Summed Area Tables (SAT)[1], is an intermediate lookup table, which stores the sum of all elements in an input matrix between the left-top corner and the current location, and allows fast computation of rectangular summation at a constant time regardless of the area of subregions. In the following sections, we first introduce the typical applications that use SAT to speed up computational performance. Next, we illustrate the motivation of optimizing the SAT on the CUDA-enabled GPUs.

6.1.1 Introduction

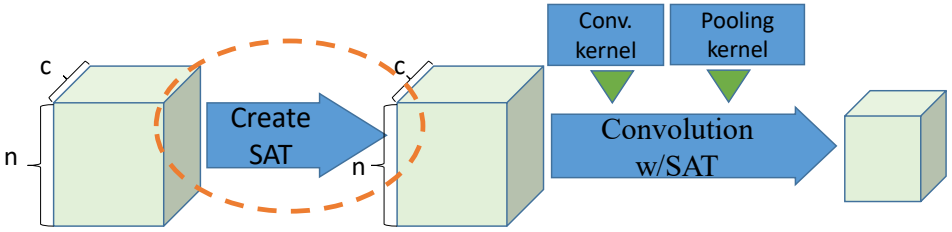
A wide range of different applications proposed and utilized algorithms for efficient SAT. For instance, it is crucial to design efficient SAT algorithms in order to accelerate the computation

[1] SAT is a 2D scan operation also called Integral Image in image processing field.

(a) Face detection [152].



(b) Objection recognition [5].



(c) Convolutional Neural Network (CNN) [68].

Fig. 6.1 SAT Applications.

pipeline in a vast range of real-time computer vision [29, 152, 52, 55, 5, 56, 17, 105, 113, 123, 143] and Deep Learning workloads [166, 42, 68] as Figure 6.1 shows. The importance of SAT makes it imperative to design SAT algorithms that would scale vertically (i.e. on one node), as well as horizontally (i.e. on the entire system). In this study, we focus on the acceleration of the SAT at the node level using a scalable method, with respect to problem size.

In this paragraph, we summarize some of the high-performance computer vision and machine learning applications that are based on the SAT to illustrate the importance of accelerating the computation of the SAT. Crow et al. [29] innovated the idea of summed area table to implement an algorithm for general texture filtering in the computer graphics field. Hensley et al. [56] applied SAT to dynamically compute image-based lighting from high-dynamic-range maps. Lewis et al. [90] successfully implemented a fast template matching algorithm based on integral imaging. Viola and Jones [152] demonstrated a real-time face detection cascade network by speeding-up the haar-liked feature computation. For visual tracking, Han [52] et al. applied an integral imaging technology to speed-up the local feature computation. Bay et al. [5] implemented an efficient and accurate object recognition algorithm based on SAT. The authors in [154] used integral imaging to speed-up a HOG-LBP human detector and achieved outstanding performance. Nehab et al. [113] presented an efficient recursive filtering implementation using summed area tables.

This section presents the motivation of optimizing SAT primitive on CUDA-enabled GPUs. In recent years, GPUs have played an important role in high-performance computing by providing superior computing capability over conventional CPUs. For some compute-intensive algorithms, GPUs can offer an order of magnitude improvement in performance compared to CPUs. However, to achieve high performance with GPUs, it is essential to optimize algorithms that meet the requirements and limitations of the complex GPU architecture. Typically there are two classes of SAT algorithms on GPUs. One is the scan-scan algorithm (and its variations). The other is the scan-transpose-scan algorithm, which adopts three or four steps to compute the SAT. The algorithms from the second class require a regular coalesced access pattern of the device global memory in one of the steps, and an expensive matrix transpose operation is applied in another step (e.g. [13, 123]).

6.1.2 Challenges & Proposed algorithms

In this section, we discuss the challenges of optimizing the SAT by SSAM and overview the proposed algorithms. *Register cache* is a promising, yet complex technique, to improve the performance of memory-bound GPU applications [34, 85, 41]. In this work, we implement a collection of efficient algorithms that compute the SAT by register cache. More specifically, we manually cache data by register files. Also, we propose a novel Block-Register-Local-Transpose methodology (BRLT) and an efficient parallelized serial scan algorithm, which is more efficient than conventional approaches, due to the reduction in inter-thread communication and arithmetic instructions. The use of register cache to compute partial prefix sums, an implicit in-register transpose and a serialized computation of the prefix sum in one dimension differentiates the algorithms proposed in this paper from the prevalent SAT algorithms on GPUs. On the contrary, the existing methods typically rely on scratchpad memory and transposition of the matrix in-between the prefix sum in each dimension.

The bottleneck of The SAT computation is data movement. Efficiently accessing global memory in a coalesced pattern is critical to the algorithm's performance. Additionally, it is essential to design algorithms that reduce the global memory access. To reduce the data movement, we rely on the register cache method since registers in GPUs provide more advantages than scratchpad memory: higher throughput, higher capacity, and lower latency. However, it is a challenge to effectively use the registers while avoiding contention. Moreover, the technology for communicating registers between threads, known as *shuffle* instruction, works only within a single warp (a set of threads executed together in CUDA). Hence, we adapt our algorithm to do the scan at the warp level while avoiding intra-warp communication. In this work, we propose three SAT computation algorithms. Using Nvidia Pascal P100 and Volta V100 GPUs, the evaluation result shows that our fastest algorithm outperforms the

state-of-the-art libraries, OpenCV and Nvidia NPP, in addition to being generic to all kinds of data types.

6.1.3 Contribution

The contributions in this chapter are as follows:

- According to our SSAM model, we implement a collection of SAT algorithms on CUDA-enabled GPUs.
- We propose a novel Block-Register-Local-Transpose (called BRLT) algorithm, which contributes to improving the dependency graph in SSAM. To our best knowledge, this is the first algorithm to apply the register cache method to compute the SAT.
- We propose another efficient SAT computation algorithm that relies on an optimized intra-thread serial scan method to reduce inter-thread communication.
- We achieved outstanding performance on computing SAT as that SSAM-based implementation outperform the production-level OpenCV and Nvidia NPP libraries using the latest Tesla P100 and V100 GPUs.

6.2 Scan operator & SAT computation

SAT computation performs the scan[2] algorithm twice on an input matrix. First, we perform rows (namely horizontal) scan on the input matrix, then we perform a scan on the columns (namely vertical) of the result matrix of the first step. Because different rows (and columns) are independent of each other, the scan operation of different rows or columns can be computed in parallel.

Summed Area Tables is an effective and quick algorithm used to compute the summation of values over an arbitrary rectangular region. Formally, the inclusive SAT is defined as

$$J(x,y) = \sum_{j=0}^{y} \sum_{i=0}^{x} I(i,j) \tag{6.1}$$

while the exclusive SAT is defined as

$$J(x,y) = \begin{cases} 0 & \text{x=0 or y=0,} \\ \sum_{j=0}^{y-1} \sum_{i=0}^{x-1} I(i,j) & \text{others} \end{cases} \tag{6.2}$$

[2]Scan is also known as all-prefix-sum: the sums of prefixes (running totals) of an input sequence

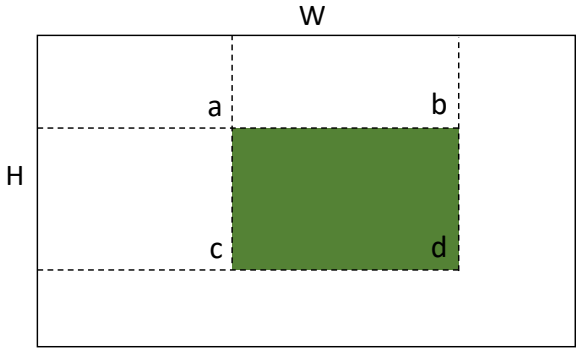


Fig. 6.2 Summed Area Table illustration.

Algorithm 1 Naive serial inclusive SAT algorithm

Input: 2D matrix I of size $H \times W$
Output: 2D matrix J of size $H \times W$

```
1: J[0][0] ← I[0][0]
2: for i = 1; i < W; i++ do
3:     J[0][i] ← I[0][i] + J[0][i − 1]
4: end for
5: for j = 1; j < H; j++ do
6:     sum ← 0
7:     for i = 0; i < W; i++ do
8:         sum ← sum + I[j][i]
9:         J[j][i] ← J[j − 1][i] + sum
10:    end for
11: end for
```

where I is an input 2D matrix of size $H \times W$. J is the output SAT matrix with the same size as I. A point in the 2D space is indicated as (x,y), where $0 \leq x < W$ and $0 \leq y < H$. The inclusive and exclusive SAT can be transformed easily to one another. In this paper, we only consider the inclusive SAT computation; the more general case of SAT. Algorithm 1 shows the naive serial inclusive SAT computation. For the input matrix of size $H \times W$, $2 * H * W$ addition operations are needed.

In Figure 6.2, the size of the SAT is H×W. The values a, b, c, and d denote the four values at the corners of the shaded rectangle. In the original matrix, the summation of the data in the shaded rectangle can be calculated efficiently as $a + d - b - c$, which needs only four accesses of the SAT lookup table and three arithmetic operations to obtain the summation of a specified rectangular area in the original 2D matrix.

6.2.1 Efficient Scan Algorithms

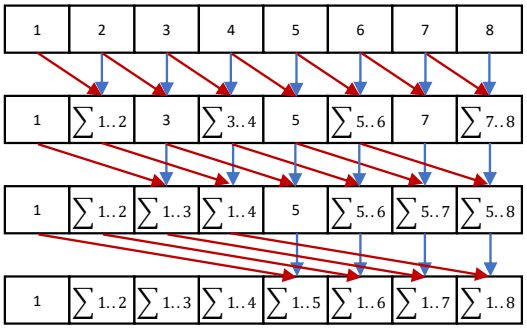


Fig. 6.3 Parallel Kogge-Stone scan for 8 elements [72].

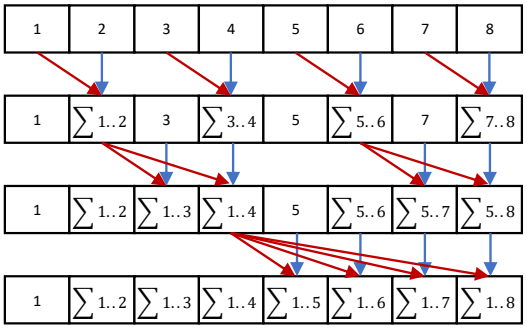


Fig. 6.4 Parallel LF-scan for 8 elements [84].

Scan (or prefix sum), a fundamental primitive, is defined as an associative and binary operator $\bigoplus$, where given a sequence of N elements $V = \{v_1, v_2, v_3 ..... v_N\}$, an inclusive scan on vector V computes a new array $U = \{u_1, u_2, u_3 ..... u_N\}$, where $u_i = \bigoplus_{j=1}^{i} u_j$. On the other hand, exclusive scan computes $u_i = \bigoplus_{j=1}^{i-1} u_j$. For simplification, scan operation is a partial sum of a sequence as introduced in literature [158]. In this study, we only focus the discussion on the inclusive SAT computation since it is the general case.

Serial Scan Algorithm As Figure 6.5 and Algorithm 4 shown, given a vector V with N elements, we require N-1 stages of computation and N-1 addition operations to compute the output array. A naive serial scan is performed by a single thread, hence the computing efficiency is very low. More specifically, regarding the scan operation, the naive serial scan is simple yet inefficient due to it can not be parallelized for a vector with a small size.

Algorithm 2 Kogge-Stone scan by a CUDA warp

Input: $data$, $laneId$ ▷ data is a register
Output: prefix sum of $data$.
1: #pragma unroll ▷ unroll loop
2: **for** $i = 1; i <= N; i* = 2$ **do**
3: $val \leftarrow shfl\_up\_sync(0xffffffff, data, i, WarpSize)$
4: **if** $laneId > i$ **then**
5: $data \leftarrow data + val$ ▷ in parallel
6: **end if**
7: **end for**

Algorithm 3 LF-scan by a CUDA warp

Input: $data$, $laneId$ ▷ data is a register
Output: prefix sum of $data$.
1: #pragma unroll ▷ unroll loop
2: **for** $i = 1; i <= N; i* = 2$ **do**
3: $val \leftarrow shfl\_sync(0xffffffff, data, i-1, i* = 2)$
4: **if** $(laneId \ \& \ ((i*2)-1))) \geq i$ **then**
5: $data \leftarrow data + val$ ▷ in parallel
6: **end if**
7: **end for**

Parallel Scan Algorithm Unlike the serial scan, parallel scan computes prefix sum in parallel. The authors in [36] implemented several kinds of CUDA-optimized scan algorithms, such as Brent-Kung pattern [19] [14], Kogge-Stone pattern [72], Han-Carlson pattern [50], and Ladner-Fischer Pattern (namely LF-scan) [84]. In theory, the LF-scan achieves the highest computing efficiency with log_2N stages and $\frac{Nlog_2N}{2}$ addition operations as Figure 6.4 shows, where N indicates the size of the input array. Kogge-Stone scan, shown in Algorithm 2, is a widely adopted algorithm. In comparison, it needs log_2N stages to compute and each stage has $S_k = N - 2^k$ addition operations, where k indicates the k^{th} stage of computation. We summarize more details of Kogge-Stone scan and LF-Scan as follows:

(i) Figure 3.3 presents the workflow of Kogge-Stone scan and Algorithm 2 shows the computation on CUDA using a single warp.

(ii) Figure 6.4 presents the workflow of LF-scan and Algorithm 3 shows the computation on CUDA using a single warp.

It is worth mentioning that the work proposed in this paper is the first to apply the LF-scan algorithm to the SAT computation workload. The serial scan is not as efficient as the parallel

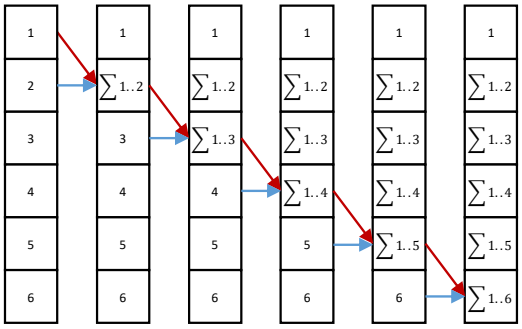


Fig. 6.5 Serial scan for 6 elements.

Algorithm 4 Serial scan to a vector

Input: 1D vector V of size N.
Output: 1D vector U of size N.

```
1: U[0] ← V[0]
2: for i = 1; i < N; i++ do
3:     U[i] ← V[i] + U[i − 1]
4: end for
```

scan algorithms to a specified one dimension array, yet the serial scan can be utilized perfectly to accelerate two-dimensional SAT computation on GPUs (as will be shown in Section 6.3.2).

6.2.2 Terminology in chapter

Throughout this paper, we define T to represent a data type. Regarding data types, 8u is an unsigned char, 32u is an unsigned int32, 32s is an int32, 32f is a float32, 64f is double. $T_A T_B$ means the input data type is T_A and the output data type is T_B. For example, 8u32u means that the input data type is 8u, after computation the output data type is 32u. For the size of matrices, W denotes width, and H is height. we use the name *warp-scan* to describe a single warp performing a scan operation.

6.3 Map SAT Computation in SSAM

This section introduces how to map the SAT algorithm in SSAM using the key techniques, e.g. register cache, partial sums computation, and communication. Except employing the efficient in-register computation, we innovate an algorithm to transpose the register matrix and thus, we improve the dependency of scan computation for better parallelism.

Using CUDA, we implement three algorithms to compute SAT efficiently. The three algorithms are listed as follows : ScanRow-BRLT, ScanRowColumn, and ScanColumn. Our algorithms compute the prefix sum (scan) twice, namely, row scan then column scan. More specifically, given an input 2D matrix I, we first compute the rows prefix sums of I to a temporary matrix L, then compute the columns prefix sums of L to output 2D matrix Y. To explain how to map SAT computation in SSAM in detail, we summary three key technologies in the proposed algorithms as follows.

Register cache We illustrate how to cache data by register files in this paragraph. Global memory access is critical to the performance of CUDA applications. We seek to minimize the data transfer from global memory as much as possible. In CUDA, all threads in a single warp execute simultaneously in a SIMT fashion. Since SAT computation is bound by memory-bandwidth, we use the registers such that each CUDA thread caches multiple words to the registers to increase data reuse. As Algorithm 5 line 1 shows, each thread uses 32 registers to cache global memory data that is declared as *T data[32]*. Note that registers cannot be used as a dynamic array, i.e. the array size residing in registers must be determined at compile time. In a single warp, a 32×32 register matrix is built up by the threads in the warp. During computation, the data can be fetched from registers without accessing global memory. By carefully designing the register cache, the global memory and caches accesses can be greatly reduced. Since the register's scope is limited to a single thread, we exchange data between threads in a warp using the CUDA shuffle instruction.

Block-Register-Local-Transpose Method We propose a novel Block-Register-Local-Transpose (BRLT) parallel algorithm to perform transposing a matrix residing in registers. We build asymmetric register matrix where each CUDA thread reserves 32 registers to cache data. Since the *WarpSize* is fixed as 32, each warp holds a register matrix of size 32×32. For each register matrix, we use an independent shared memory matrix of size 32×33 as a temporary buffer to transpose the register matrix. As shown in Algorithm 5, we reserve S blocks in shared memory depending on the number of batches in a single transposition and with consideration to the available capacity of shared memory. For instance, the size of shared memory on Tesla P100 GPUs is 64KB (as listed in Table 1.1) and the maximum block size (BlockSize=1024) is used to achieve the highest occupancy if the data type T is 32f or 32s. To avoid register pressure we use a block size (BlockSize=512) instead when T is double. The S can then be computed as *S=32/sizeof(T)*. This indicates that S varies between 4 and 8 according to the data type of T. It is worth mentioning that we avoid shared memory bank conflicts by setting the *sMem* stride to the size 33 rather than 32 (shown in line 2 in

Algorithm 5), due to the shared memory is divided into 32 banks, the mechanism is similar to the banks in Dynamic Random Access Memory (DRAM) and each bank responses only one request each time. A simple application of the BTLT method can be found in later Figure 6.7.

Algorithm 5 Block-Register-Local-Transpose (BRLT)

```
1:  T data[32];                                          ▷ an array of 32 registers
2:  __shared__ T sMem[S][32][33]                         ▷ S is tuned to the shared memory size
3:  for i = 0; i < WarpCount; i+ = S do
4:      if i ⩽ warpId < i + S then
5:          k ← warpId − i
6:          #pragma unroll
7:          for j = 0; j < 32; j++ do
8:              sMem[k][j][laneId] = data[j]             ▷ in parallel
9:          end for
10:         #pragma unroll
11:         for j = 0; j < 32; j++ do
12:             data[j] = sMem[k][laneId][j]             ▷ in parallel
13:         end for
14:     end if
15:     __syncthreads();
16: end for
```

In Algorithm 5, the variable of data represents registers rather than an array and thus, it is necessary to use data as registers explicitly and unroll the loops in compiling time, i.e. we use "#pragma unroll" instruction in both line 7 and line 11. The overhead of such a matrix transpose is very lightweight, as will be proven by our performance model in Section 6.4. It is noteworthy that all 32×32 register matrixes are processed in parallel.

Partial Sum Computation This paragraph presents the partial sums computation for Scan operator in a CUDA Block. There are two levels of the computation of the partial sum: warp level and block level. Algorithm 4, Algorithm 2, and Algorithm 3 illustrate three kinds of scan algorithms for the computation of the partial sum using a single warp. It is very important to note that all these algorithms can be parallelized on GPUs. The block-level partial sum computation will be presented in later sections.

In Figure 6.8, we illustrate how to compute the partial prefix sum of each warp using the shared memory. There are three steps. Step #1 we store the partial sum (last row of the register matrix) from registers to shared memory according to their *warpId*: in total, we populate a $WarpCount \times WarpSize$ matrix on shared memory. Step #2 we compute the prefix

sum of data which is stored in shared memory. Finally, step #3, in each block, each warp fetches the specified values from shared memory and accumulates them to the corresponding registers.

6.3.1 SSAM-based ScanRow-BRLT Method

In this section, we introduce the SSAM-based ScanRow-BRLT algorithm. SSAM-based ScanRow-BRLT algorithm is the improvement of the scan-transpose-scan SAT algorithm as in literature [13]. The original scan-transpose-scan SAT algorithm saves the row scan result to global memory and executes a transposing operation on global memory explicitly. It is clear that the cost of writing and reading data via global memory is very heavy since the lowest throughput of global memory as introduce in Section 2.1.3. On the contrary, in our SSAM-based ScanRow-BRLT algorithm we use register cache to perform the parallel warp-scan and apply BRLT to the prefix sum result. Hence, we can fully take advantage of the register to perform the in-register computations. More specifically, before the row scan data is stored in global memory, the BRLT is performed. Finally, the stored result in global memory is the row prefix sum result that is already transposed.

In the ScanRow algorithm, the warp-scan is crucial to the performance of SAT computation and thus, we employ the shuffle based warp scan to avoid using shared memory as in Algorithm 2 and Algorithm 3. In other words, we employ the shuffle intrinsic to perform the intra-warp communication (transfer the partial sums).

6.3.2 SSAM-based BRLT-ScanRow Method

In this section, we present the SSAM-based BRLT-ScanRow algorithm. As Figure 6.6, 6.7, 6.8 shown, the BRLT-ScanRow method computes the prefix sum of all rows and transposes the input matrix in the same step, without using the global memory as a temporary buffer (as in [13][123]). The input 2D matrix is divided along the column as Fig 6.6 shows. For instance, given sizeof(T) is 4, we would use

$$\begin{cases} BlockDim.x = 1024 \\ BlockDim.y = 1 \\ BlockDim.z = 1 \end{cases}$$

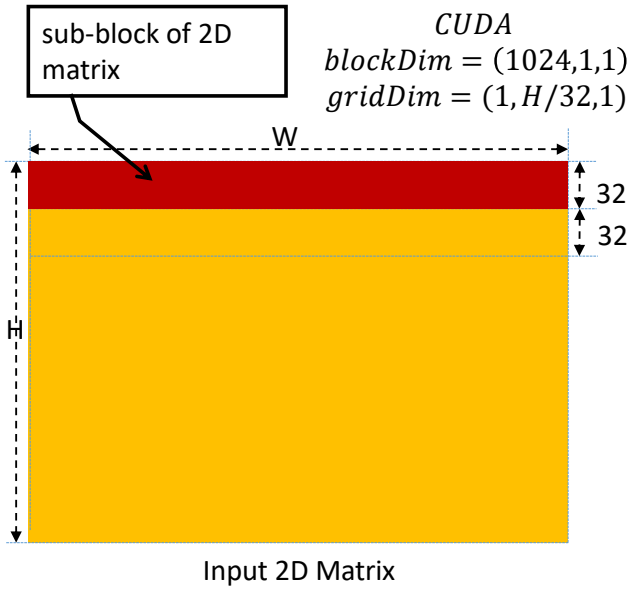


Fig. 6.6 BRLT-ScanRow algorithm : mapping CUDA grids and blocks to the input matrix.

$$\begin{cases} GridDim.x = 1 \\ GridDim.y = \lceil \dfrac{H}{WarpCount} \rceil \\ GridDim.z = 1 \end{cases}$$

In Figure 6.6, we illustrate how to map CUDA grids and blocks to the input 2D matrix, the detailed computation is as uppon formulations. As Fig 6.7 shown, first we build a register matrix of size $32 \times WarpSize$ for each warp (each thread holds 32 elements of data using registers). Next, we apply the BRLT method to transpose the register matrix in each CUDA block. Then, we apply a serial scan in each CUDA thread to compute the prefix sum. After that, as Fig 6.8c shows, the partial prefix sum of each warp is stored to the shared memory of size $WarpCount \times WarpSize$ and compute prefix sum in the shared memory. Finally, the data stored in the shared memory is parallelly aggregated to all the values held in the register cache in their respective $warpId$.

In Figure 6.8, we present how to perform inter-warp communication by shared memory. Since the size of a single Warp is very limited and thus, the size of the processed data by a warp at each time is restricted as 32×32. Hence, using inter-warp communication, the computational efficiency of scan operation can be improved. As the step #1 shown in Figure 6.8, the partial sums which are held by all warps in a CUDA block are stored to shared memory. Next, in step #2, we perform the serial on the shared memory. Then, in step #3, the

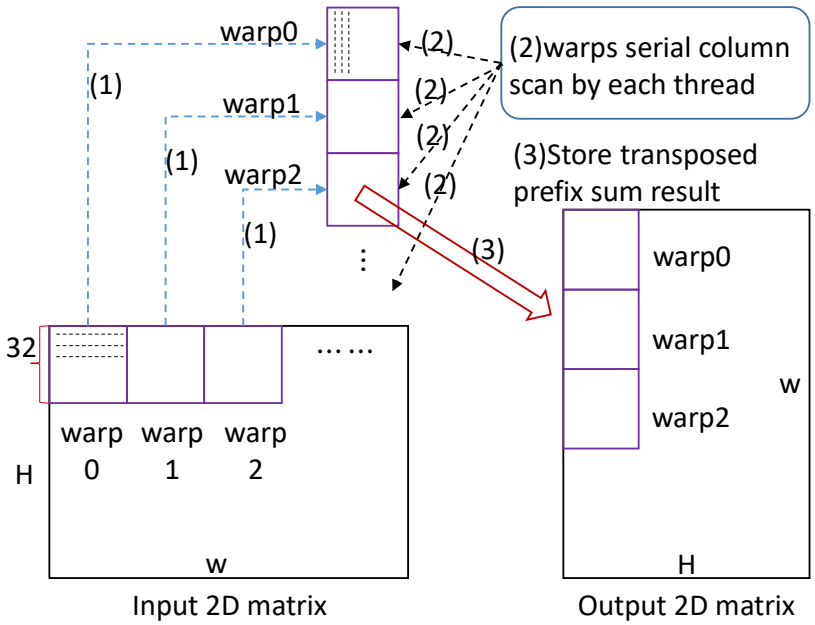


Fig. 6.7 BRLT-ScanRow algorithm: illustration of how to use the BRLT algorithm and serial scan.

accumulated partial results in shared memory are accumulated back to the 32×32 register matrix which is held by all warps again. In all of the uppon steps, all threads perform in parallel without any inter-thread communication. The advance of such kind of serial scan can be justified by the evaluated result. Furthermore, we build a performance model to elucidate its mechanism as will be shown in Section 6.4. Also, the 32×32 shared memory we used in Figure 6.8 is the same shared memory buffer that is used in the BRLT operation. In other words, the same shared memory is used for both BRLT operation and the inter-warp communication due to they perform in sequential order without any conflicts of operations.

It is noteworthy that we simply repeat the BRLT-ScanRow process twice, where the first output matrix becomes the input of the second computation, and the final result is the required SAT result. Though both ScanRow-BRLT and BRLT-ScanRow perform the same scan operation to the input matrix, the biggest difference between them is the order of operations. In ScanRow-BRLT, we firstly perform scan operation, then use the BRLT algorithm. However, in BRLT-ScanRow, we first perform the BRLT operation then execute the scan operation.

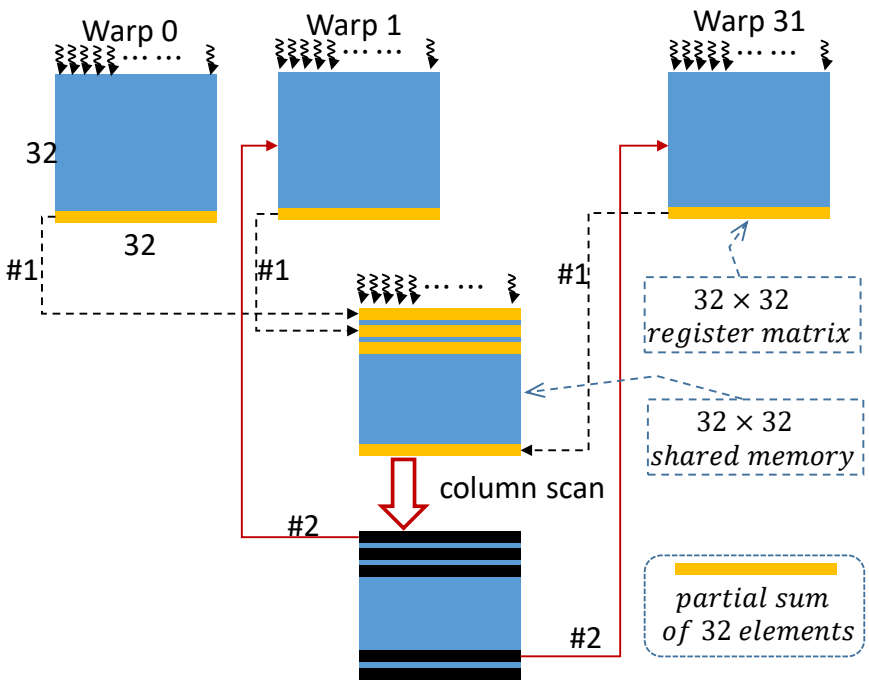


Fig. 6.8 BRLT-ScanRow algorithm: illustration of using shared memory in the intra-block communication and accumulating the partial-sum of each warp.

6.3.3 SSAM-based ScanRowColumn Method

This section introduces the ScanRowColumn algorithm. Similar to the BRLT-optimized methods in the earlier sections, ScanRowColumn also uses the key techniques in SSAM, e.g. register cache, shuffle intrinsic, shared memory, etc. However, in ScanRowColumn algorithm, we avoid using the BRLT method, which can be seen as the overhead of the SAT computation. To be simple, ScanRowColumn algorithm performs the SAT computation without using BRLT method.

SSAM-based ScanRow Method SSAM-based ScanRow method is used to compute the prefix sum of rows (horizontal direction) for the input matrix as Fig 6.6 shows. The input matrix is divided into multiple successive blocks along the column. Each CUDA block computes a subset of the input matrix of size WarpCount$\times$W. To avoid intra-block communication, each CUDA warp caches and processes successive data in a row of the input matrix. Because each thread caches C elements (C=*32* is used), each warp can cache and process data of size *WarpSize*C=1024* elements at each step. If the input matrix width is bigger than 1024, the warps repeat the same process to the data after 1024.

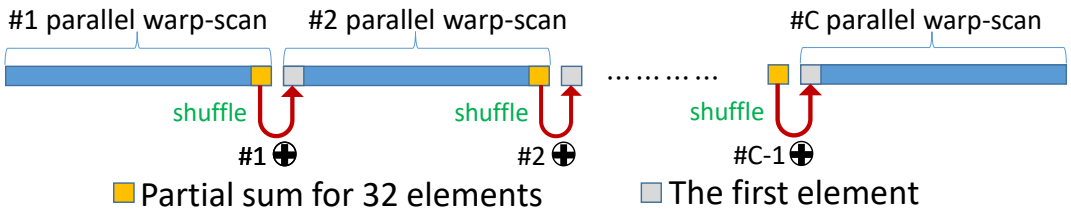


Fig. 6.9 Using ScanRow to compute the prefix sum of a row in the input 2D matrix by a single warp.

As Figure 6.9 shows, a parallel warp-scan algorithm is applied to the first 32 elements as step *#1*, the last partial sum is added to the first element of the next 32 elements (the shuffle instruction is used for intra-warp communication), then we repeat the same parallel warp-scan to the elements as shown in step *#2*. After repeating this process C times, a row of prefix sum results of size *C*WarpSize* can be obtained. We use the following values when launching our CUDA kernel:

$$\begin{cases} BlockDim.x = 1024 * \dfrac{sizeof(float)}{sizeof(T)} \\ BlockDim.y = 1 \\ BlockDim.z = 1 \end{cases}$$

$$\begin{cases} GridDim.x = 1 \\ GridDim.y = \lceil \dfrac{H}{WarpCount} \rceil \\ GridDim.z = 1 \end{cases}$$

In Figure 6.9, we illustrate how to perform ScanRow computation by a single Warp and how to accumulate the partial sums. It is very important to point out that before performing a parallel warp scan in the next step, the current partial sums are sent to the first thread via shuffle intrinsic directly and accumulate the partial sums in the first element in an array of size 32 (namely WarpSize) only.

SSAM-based ScanColumn Method Similar to Section 6.3.3, the input matrix is divided into multiple blocks along the row (horizontal direction). The SSAM-based ScanColumn method is used to compute the prefix sum along the columns by serial warp-scan method rather than parallel warp-scan. Because serial warp-scan performs much better in the column direction, the detailed discussion is reserved to a later section (Section 6.4.1). Each thread caches C elements of data, then each warp builds a register matrix of size $C \times WarpSize$

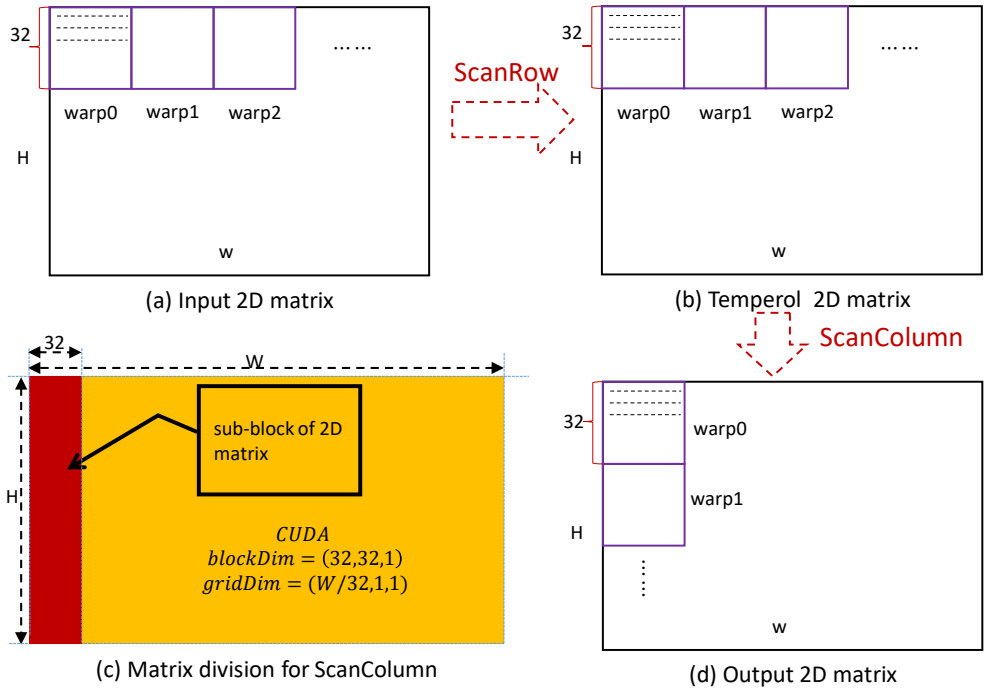


Fig. 6.10 Illustration of the ScanRowColumn algorithm by ScanRow and ScanColumn operations.

(namely 32×32) at each step. For each CUDA block, at each step, the prefix sum of size $BlockSize \times C$ can be computed. When the H (namely height) of the input matrix is larger than $BlockSize$, this process will be repeated for all the rows. We use the following values when launching our CUDA kernel:

$$\begin{cases} BlockDim.x = 32 \\ BlockDim.y = 32 \\ BlockDim.z = 1 \end{cases}$$

$$\begin{cases} GridDim.x = \lceil \frac{W}{C} \rceil \\ GridDim.y = 1 \\ GridDim.z = 1 \end{cases}$$

As Figure 6.10 shown, to be simple, first we use the ScanRow algorithm to process the input matrix where the CUDA block is mapped as in Figure 6.6; Next, we use ScanColumn to process the output from the previous step where the CUDA block is mapped as the figure in the bottom left.

6.4 Performance Model

We analyze the benefit of using a serial scan on each CUDA thread: a main overhead in the BRLT algorithm. We prove that this overhead is negligible by analyzing its effects. Additionally, based on the architecture details of GPU, we give a deep insight into the performance-improving details of our algorithms. It is noteworthy that we rely on micro-benchmarking to better understand the characteristics of GPUs and the measured values are listed in Section 4.4.1.

6.4.1 Single Warp and Single 32×32 Register Matrix

Our application's optimization strategy is to use a single warp to process a register matrix of size 32×32 (namely $C \times WarpSize$, $C = 32$) at each step. We apply three kinds of methods to compute the prefix sum in the column or row directions. Since the latency of registers is as small as 1 clock, we ignore its effect on performance and focus on discussing the shared memory access and arithmetic operations.

Transpose Operation To locally transpose the register matrix of size 32×32 efficiently, we use the shared memory of size 32×33 as a temporary buffer (Algorithm 5) by moving the data from the registers to the shared memory row by row before loading the data from shared memory column to column again. The total number of accesses shared memory (both load and store) is

$$N_{trans\_store\_smem} = N_{trans\_load\_smem} = C * WarpSize$$

It is very clear that $N_{trans\_store\_smem} = N_{trans\_load\_smem} = 1024$. The number of required operation stages is

$$N_{scan\_row\_stage} = C + C$$

We can obtain that $N_{scan\_row\_stage}$=64 since C is 32 as introduced earlier. Based on the shared memory latency we have measured, the latency of transposing a register matrix of size 32×32 can be estimated as

$$L_{transpose} = N_{scan\_row\_stage} * T_{smem} \tag{6.3}$$

Where T_{smem} is the latency of shared memory access as presented in Section 4.4.1.

Scan Row Operation Using parallel warp-scan to compute the prefix sum of 32×32 registers in-place, at least the number of required stages (e.g. Kogge-Stone scan and LF-scan) maybe written as

$$N_{scan\_row\_stage} = log_2(WarpSize) * C$$

We can derive that $N_{scan\_row\_stage}$=160. The number of addition operations in Kogge-Sone scan is

$$N_{Kogge-Stone\_add} = C * \sum_{i=0}^{log_2(WarpSize)} KS\_add_i$$

Where, KS_add_i is defined as the number of addition operations in each stages of Algorithm 2. Obviously, the $log_2(WarpSize)$ is constant as 5 and KS_add_i in uppon equation can be counted in Algorithm 2. To be simple, $N_{Kogge-Stone\_add} = (31+30+28+24+16) * C = 4128$. The number of the shuffle instructions in both parallel warp-scan (Algorithm 2 and Algorithm 3) in a warp is

$$N_{scan\_row\_sfl} = N_{scan\_row\_stage}$$

namely, $N_{scan\_row\_sfl}$=160. The number of addition operations in LF-scan is

$$N_{LF\_add} = C * \sum_{i=0}^{log_2(WarpSize)} L\_add_i$$

Where, L_add_i is defined as the number of addition operations in each stages of Algorithm 3. Simillary to the calculation of $N_{Kogge-Stone\_add}$, the L_add_i can be counted in Algorithm 3. For simplication, $N_{LF\_add} = (16+16+16+16+16) * C = 2560$. In a scan row algorithm the latency that comes from shuffle instructions and addition operations is

$$L_{scan\_row} = N_{scan\_row\_stage} * (T_{shuffle} + T_{ADD}) \tag{6.4}$$

However, additional boolean AND operations are needed as shown in Algorithm 3 line 4

$$N_{LF\_and} = (WarpSize * N_{scan\_row\_stage}) * C$$

We can derive that N_{LF_and} is as constant as 5120.

Scan Column Operation In each warp, we compute the prefix sum in the column direction using the serial warp-scan method (Algorithm 4). The number of needed stages is

$$N_{scan\_col\_stage} = C - 1$$

It is notable that $N_{scan\_col\_stage}$ is a constant value, namely 31. Since each thread in a warp executes the serial warp-scan without both any intra-warp communication and thread divergence, the total number of concurrent addition operations is

$$N_{scan\_col\_add} = WarpSize * N_{scan\_col\_stage}$$

We can easily obtain that $N_{scan\_col\_add}$ is const as 992. The latency of scan column can be estimated as

$$L_{scan\_col} = N_{scan\_col\_stage} * T_{ADD} \tag{6.5}$$

6.4.2 Analyzing Latency and Throughput

We analyze the latency and throughput of compute a 32×32 matrix in this section. On one hand, based on the Equation 6.3∼6.5, we can find out that

$$L_{transpose} + L_{scan\_col} \ll L_{scan\_row} \tag{6.6}$$

It means the latency of transposing the register matrix and serial scan for column is very low. On the other hand, we consider the aggregate throughput for warps in a CUDA kernel. The number of active warps can be computed as

$$N_{wpb} = \lfloor \frac{BlockSize}{WarpSize} \rfloor \tag{6.7}$$

$$\begin{aligned} N_{active\_warps} = N_{sm} * min(\lfloor \frac{Reg\_sm}{Reg\_thread * WarpSize} \rfloor, \\ \lfloor \frac{Smem\_sm}{Smem\_block} \rfloor * N_{wpb}, N_{wpb} * N_{max\_blk\_sm}); \end{aligned} \tag{6.8}$$

where N_{wpb} denotes warp count per SMs, N_{active_warps} is the total warp count of our CUDA kernel, N_{sm} is the total count of SMs, *Smem_sm* is the available shared memory size per SMs, *Smem_block* is the used shared memory in our CUDA kernel, and $N_{max\_blk\_sm}$ is the maximum available block count per SM. The shared memory bandwidth for each active warp can be expressed as

$$BW_{warp} = \frac{BW_{Smem}}{N_{active\_warps}} \tag{6.9}$$

where BW_{Smem} is the shared memory bandwidth. In our CUDA kernel, the time required to transpose a register matrix of size 32×32 is

$$T_{trans} = \frac{(N_{trans\_store\_smem} + N_{trans\_load\_smem}) * sizeof(T)}{BW_{Smem}} \tag{6.10}$$

Using serial warp-scan, the time of arithmetic operations may be written as

$$T_{scan\_col\_add} = \frac{N_{scan\_col\_add}}{BW_{add}} \tag{6.11}$$

Using parallel warp-scan, the required time for shuffle instructions can be expressed as

$$T_{shuffle} = \frac{N_{scan\_row\_sfl}}{BW_{shuffle}} \tag{6.12}$$

where $BW_{shuffle}$ is the shuffle instruction throughput. The required time for the arithmetic pipeline varies from method to another. Take the Kogge-Stone scan for example

$$T_{Kogge-Stone\_add} = \frac{N_{Kogge-Stone\_add}}{BW_{add}} \tag{6.13}$$

where BW_{add} indicates addition operations throughput. We can easily derive that

$$T_{Kogge-Stone\_add} + T_{shuffle} \gg T_{trans} + T_{scan\_col\_add} \tag{6.14}$$

Similarly, we can obtain the function also as

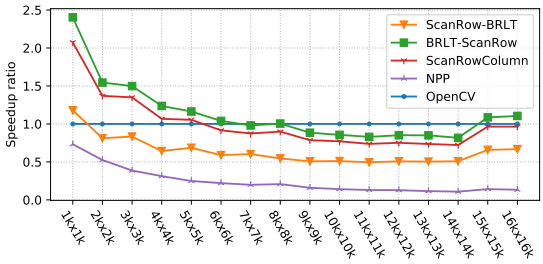
$$T_{LF\_add} + T_{LF\_and} + T_{shuffle} \gg T_{trans} + T_{scan\_col\_add} \tag{6.15}$$

Note that to Equation 6.14 and Equation 6.15, we suppose the shared memory access achieves the peak of memory bandwidth without bank conflicts. Otherwise, the shared memory throughput will dramatically degrade, and our model would have to be adjusted accordingly. Finally, the bandwidth of shared memory we use in our model is 9519GB/s on P100 and 13800GB/s on V100 as reported by [63], respectively.

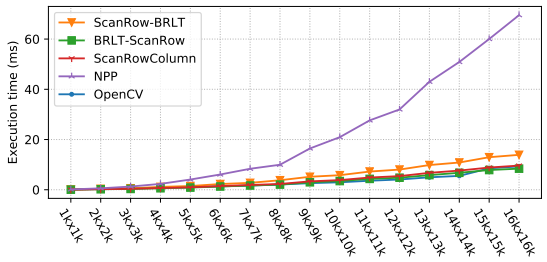
To sum up, we prove the overhead of BRLT is insignificant. Furthermore, due to the utilization of BRLT algorithm, the serial scan performs greatly better than the parallel scan (i.e. Kogge-Stone scan and LF-scan) due to the efficient in-register scan operation for the 32×32 matrix.

6.5 Evaluation

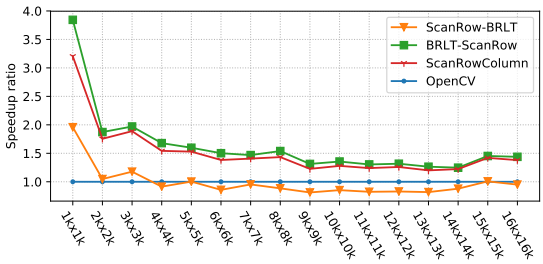
We report the performance of the proposed algorithms and discuss comparisons with NPP and OpenCV libraries. It is noteworthy that both of the two libraries will be introduced in a later section.



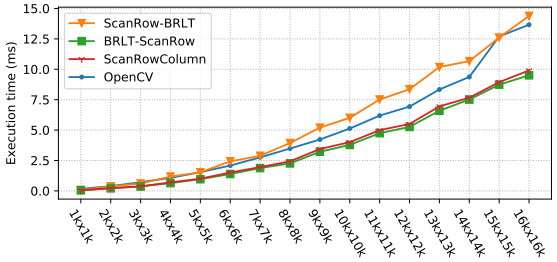
(a) Speedup of 8u32s on Tesla P100 GPU.



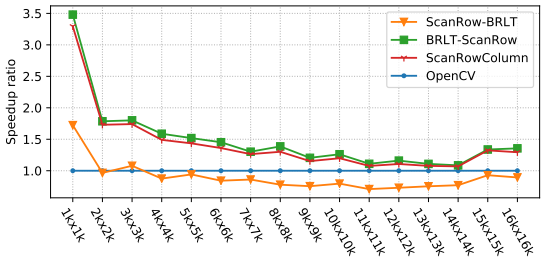
(b) Execution time of 8u32s on Tesla P100 GPU.



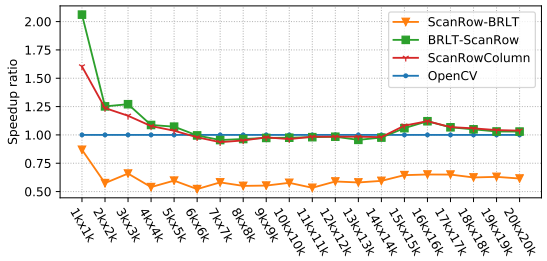
(c) Speedup of 32f32f on Tesla P100 GPU.



(d) Execution time of 32f32f on Tesla P100 GPU.



(e) Speedup of 32u32u on Tesla P100 GPU.

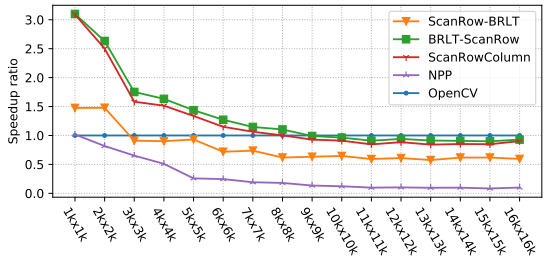


(f) Speedup of 64f64f on Tesla P100 GPU.

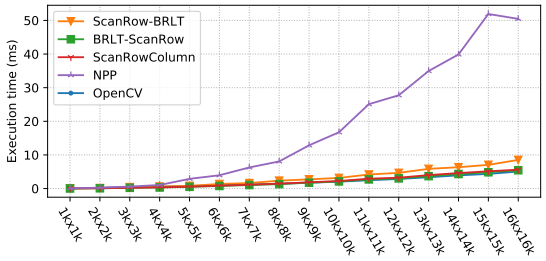
Fig. 6.11 Speedup (we use OpenCV's implementation as the baseline) and execution time of SAT on a single Tesla P100 GPU.

6.5.1 Experimental Setup

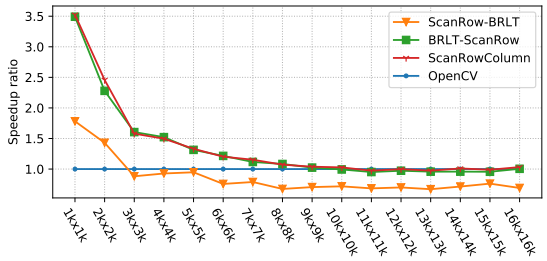
Our algorithms are implemented by Nvidia CUDA Toolkit 9.0. To obtain the highest accuracy of CUDA kernel execution time (often tens of *us*), the *nvprof* with print-gpu-trace option



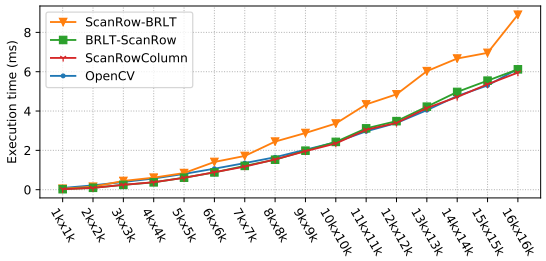
(a) Speedup of 8u32s on Tesla V100 GPU.



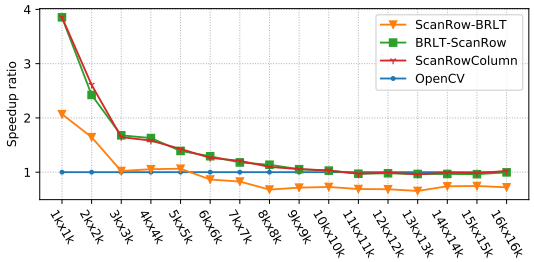
(b) Execution time of 8u32s on Tesla V100 GPU.



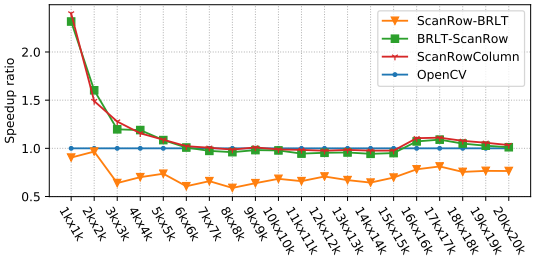
(c) Speedup of 32f32f on Tesla V100 GPU.



(d) Execution time of 32f32f on Tesla V100 GPU.



(e) Speedup of 32u32u on Tesla V100 GPU.



(f) Speedup of 64f64f on Tesla V100 GPU.

Fig. 6.12 Speedup (we use OpenCV's implementation as the baseline) and execution time of SAT on a single Tesla V100 GPU.

is used to observe the CUDA kernel execution. SAT functions that are implemented by the OpenCV 3.4.1 [18] and Nvidia NPP 9.0 [121] libraries are adopted for performance comparison. Nvidia Tesla P100 and V100 GPUs are used for performance evaluation and the ECC option is ON. Different sizes ($1k \times 1k \sim 16k \times 16k$) and data types are evaluated, e.g. 8u, 32u, 32f and 64f. It is worth mentioning that precision loss and overflow can happen in scan operators. It depends on the problem size, data type, and the input values. It is outside the scope of this work to address such a problem on a case-by-case basis (as in [55, 138]).

Table 6.1 NPP CUDA kernel details

kernel	blockSize	gridSize	Regs	SSMem	DSMem
scanRow	$(256,1,1)$	$(1,H,1)$	20	2.25KB	0
scanCol	$(1,256,1)$	$(W+1,1,1)$	18	2.25KB	0

6.5.2 Performance Overview

Both our algorithms and OpenCV implement the SAT by C++ template functions, so we can compare performance with OpenCV for all kinds of data types. In our experiments, we evaluate both the Kogge-Stone scan and LF-scan. However, they achieve nearly the same computing efficiency in our implementation. To simplify, in Figure 6.11 and Figure 6.12, Kogge-Stone method is applied for parallel warp-scan to compute all of the results. For some data types, the performance graphic nearly the same, so we report one of them: e.g. 8u32s, 8u32u, and 8u32f. When comparing the execution time with other libraries, we report the execution time for different data types, such as in Figure 6.11b, Figure 6.11d, Figure 6.12b and Figure 6.12d.

Nvidia NPP The NVIDIA Performance Primitives (NPP) is an optimized library for performing CUDA accelerated computations. The SAT function provided by *NPP* is *nppIntegral*. However, only two kinds of input and output data type are provided by NPP: 8u32s and 8u32f. Considering that NPP is closed-source, we investigated its binary by the *nvprof* and *cuobjdump* tools, which are provided by Nvidia. The investigated results are listed in Table 6.1. *Regs* is the number of registers used per CUDA thread. *SSMem* and *DSMem* are the static and dynamic shared memory allocated per CUDA block, respectively.

OpenCV Library OpenCV (Open Source Computer Vision Library) is a highly optimized library designed for efficient computation and real-time applications. We use the latest version v3.4.1 for our evaluation. The SAT function *integral()* is implemented by OpenCV using the scan-scan algorithm, their functions are *vertical_pass()* and *horisontal_pass()*. Additionally, using *shuffle* instruction, OpenCV specifically optimizes *8u* (namely *unsigned char*) input data for the horizontal scan, the function name is *horisontal_pass_8u_shfl*. It is worth mentioning that *horisontal_pass_8u_shfl* employs two optimization strategies, one of them is using the SSAM-based parallel Kogge-Stone scan algorithm, the other is that each thread loads 16 *bytes* data to registers using *uint4* data type, then applies a serial-scan to these 16 elements on each threads. However, *horisontal_pass_8u_shfl* is customized and works only for *8u* input data.

6.5.3 Speedup

Figure 6.11 and Figure 6.12 show several comparison results. The execution times are measured by *nvprof*. We disregard the data copy time between host and device since it is the same for all libraries. We manually accumulate the two compute kernels (namely row and column prefix sum) and average 10 executions. Register pressure results in the speedup disappear when matrices go to larger. Even though we evaluated matrix sizes up to 20k×20k, in real applications, the matrix sizes are typically smaller than 8k×8k (as in [123, 113, 110, 40, 127]). On a single Tesla P100 GPU and V100 GPU, our best algorithm performs up to 2.3× faster than OpenCV, and 3.2× than NPP.

Kogge-stone Scan and LF-scan In our experiments, both Kogge-stone scan and LF-scan are evaluated. Since the SAT computation is memory-bound, we can not gain speedup from LF-scan as in [36].

Generality & Simplicity Our fastest algorithm, namely BRLT-ScanRow, is implemented as a single CUDA kernel and repeatedly called twice to compute an SAT. It is simple to be implemented and general to all of the data types. However, NPP, OpenCV and other implementations [13, 32] use multiple CUDA kernels, often customized to different data types.

32f32f for Deep Learning As discussed earlier, OpenCV's implementation varies from one data type to another. For image processing computation, the data type 8u is widely used to represent the image data. However, in Deep Learning computation, 32f is the prevalent data type. Our 32f32f outperform OpenCV as shown in Figure 6.11c and Figure 6.12c.

64f64f Figure 6.11f and Figure 6.12f shows that our implementation performs better than OpenCV for up to image matrix size $6k \times 6k$.

6.5.4 Model Verification on BRLT Overhead

In this section, we analyze the BRLT overhead by displaying the breakdown of the SSAM-based SAT computations, and justify the validity of our performance model as introduced in Section 6.4.

In Figure 6.13, we display the detailed execution time of the two steps of computation. Since the BRLT algorithm is used to improve the SSAM-based SAT computations, the BRLT-ScanRow and ScanRow-BRLT function is called twice. ScanRow and ScanColumn

are used once to compute the prefix sum of rows and columns, respectively. In this section, T denotes the execution time. We can verify our performance model as

1. $T_{ScanColumn} < T_{BRLT-ScanRow}$. After BRLT finishes, ScanColumn and BRLT-ScanRow compute the column prefix sum similarly, so we can easily derive that BRLT is the overhead operation.

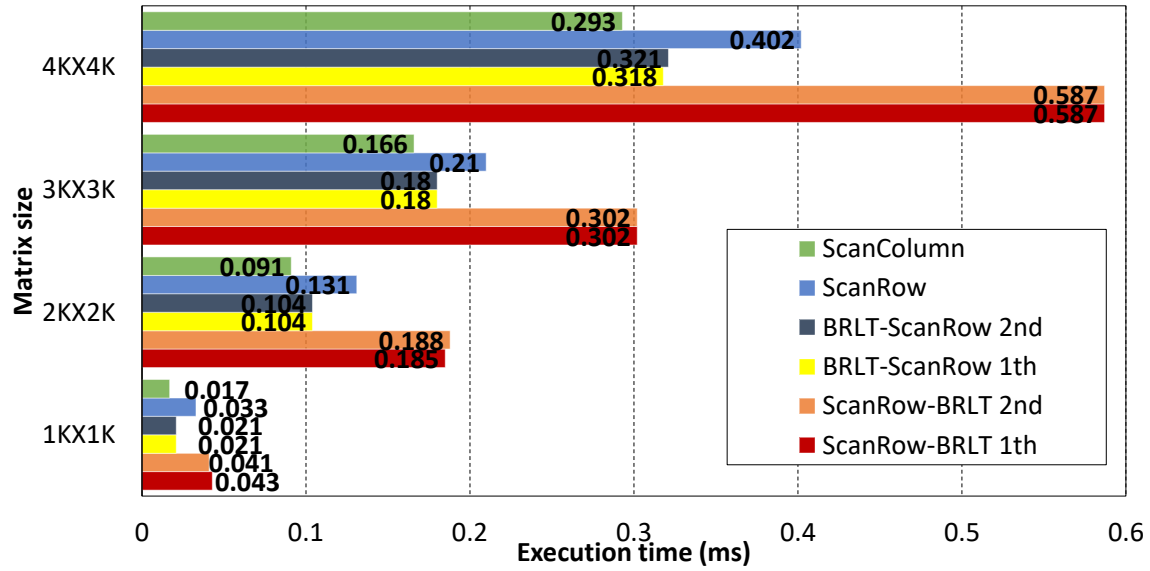
2. $2 * T_{BRLT-ScanRow} < T_{ScanRow} + T_{ScanColumn}$ indicates some benefits from BRLT operation in computing SAT.

3. $T_{BRLT-ScanRow} > T_{ScanRow-BRLT}$ means that our serial warp-scan(Algorithm 4) method is more efficient than shuffle-based parallel warp-scan(Algorithm 2, Algorithm3).

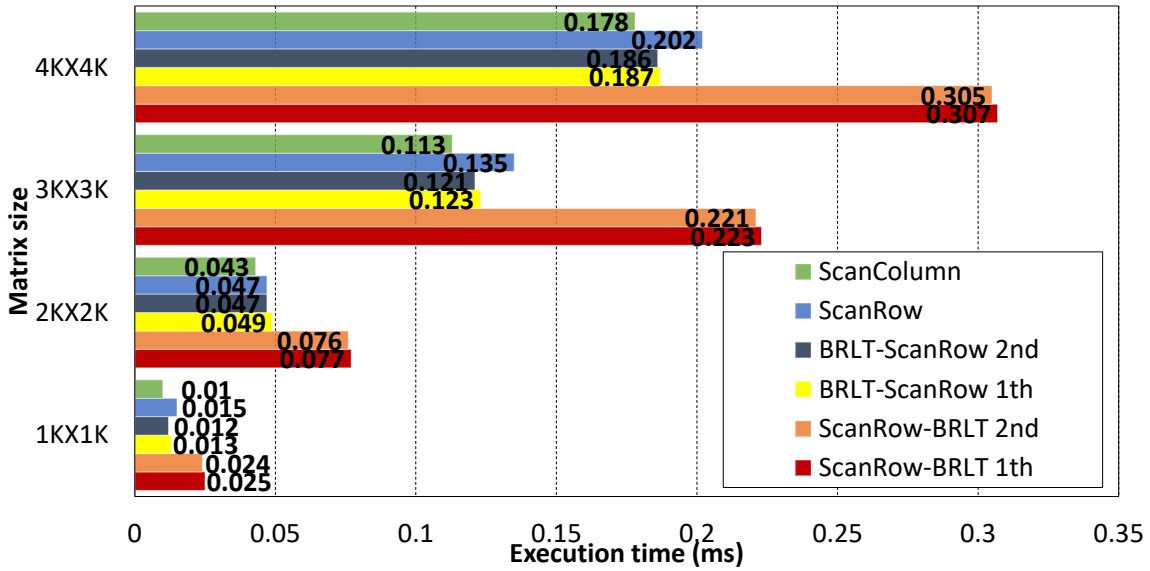
6.6 Summary

SAT is a widely used primitive in many applications. In the past two decades, the SAT successfully accelerated many image filtering and local feature extracting algorithms, and recently boosted many deep learning workloads also.

In terms of hardware architecture and computing efficiency, it is challenging to optimize the SAT implementation on GPUs. Based on our SSAM algorithm (e.g. register cache technology), we implemented three kinds of algorithms for fast SAT computation on CUDA-enabled GPUs. With our novel BRLT method that can improve the dependency of SSAM, we can transform the scan problem from horizontal operations to vertical, which results in reducing both the arithmetic computation and communication between threads, while achieving coalesced access to global memory. Furthermore, the BRLT overhead is shown to be negligible. We provide a deep insight to register cache and shared memory cache for programming CUDA-enabled GPUs. The BRLT method is general and can be applied to optimize many other algorithms, such as Fast Fourier Transform (FFT), Wavelet Transform, Discrete Cosine Transform (DCT), etc.



(a) 32f32f evaluation on Tesla P100 GPU



(b) 32f32f evaluation on Tesla V100 GPU

Fig. 6.13 Performance breakdown of computing 32f32f ranging from $1k \times 1k$ to $4k \times 4k$ input matrices. For the specified input matrix, the 1^{st} and 2^{nd} computing time are displayed: namely 1^{st} scan and 2^{nd} scan.

Chapter 7

Related Work

In this chapter, we discuss the related work. Firstly, we introduce the concept of register cache and list literature that uses register cache methodology. Secondly, we outline the register optimization research on GPUs. Thirdly, we display the research on optimizing stencil kernels. Fourthly, we discuss the partial sums methodology in the related work. Finally, we summarize related work.

7.1 Register cache

Register cache, an application optimization technique on GPUs, was named by Ben-Sasson et al. [8]. The motivations of using registers as a cache are as simple as that registers are the fastest on-chip memory with very huge capacity as explained in Table 1.1. In fact, Register cache methods are widely used on GPUs to boost individual application's performance [85, 41, 59, 8] with a variant of names.

Most notably, a technology called *register packing* was presented by Davidson et al. [34] on how to avoid shared memory communication for improving Cyclic Reduction performance. More specifically, the authors employed register packing technique to deal with the Cyclic Reduction (a tridiagonal solver with downsweep pattern) without using shared communication on GTX 460 GPU and achieved outstanding performance as that: 3$\sim$4.5 and 1.5$\sim$3 faster than the original CR implementation and other GPU tridiagonal solvers, respectively.

Lai and Seznec [85] suggested a method named *register blocking* to explore the potential peak performance of SGEMM on Fermi and Kepler GPUs. To estimate GPU applications' performance upper bound, the authors presented an analytical performance model with assembly code level benchmarking. We display the reported performance as follows: the optimized SGEMM implementations achieve about 5% better performance than CUBLAS

for large matrices on GTX580. The achieved performance is about 90% of the estimated upper-bound performance on GTX580 GPU. On GTX680 GPU, the achieved performance is about 77.3% of the estimated performance upper bound.

Enfedaque et al. [41] proposed a register-based implementation to optimize a Discrete Wavelet Transform (DWT) algorithm for fast image processing. Due to the heavy improvement of reuse data, such a programming strategy can significantly improve the performance of applications. The experimental results denote that the register-based method is more than $4\times$ faster than the best GPU implementation of the DWT.

The binary finite field multiplication algorithm was implemented by Eli Ben-Sasson et al. [8] yielded up to $138\times$ speedup than the popular Number Theory Library. It is very important to note that the key to efficient implementation is a novel performance optimization methodology, named as register cache by the authors. The authors presented how to using shuffle intrinsic to perform intra-warp communication for replacing shared memory utilization. In other words, Eli Ben-Sasson et al. developed the register cache abstraction on GPUs and illustrated how to use it in their targeting applications. Also, using the 1-D stencil as a simple example, Eli Ben-Sasson et al. illustrate how register cache and shuffle instruction works.

In literature [59], Hou et al. presented a registered-based sort algorithm that adaptively combines or splits segments of different sizes for load-balanced operations on GPUs. The experimental results demonstrate that the register-based sort algorithm performs greatly faster than existing methods on NVIDIA K80-Kepler and TitanX-Pascal GPUs.

7.2 Register optimization

Register optimization is an important approach to improve the performance of a variant of applications due to the benefit in many aspects: increasing the ILP, alleviating register pressure, tuning register allocation and instruction scheduling.

In [132], Rawat et al. present a register optimization framework for multi-statement and high-order stencil stencils. Such complex stencils are treated as a collection of trees with significant data reuse across nodes and achieved performance gain by reducing register pressure and improving register allocation. Note that register pressure is reduced by decreasing the simultaneously live ranges of nodes. In addition, a set of optimized stencils are evaluated in Section 5.3.1 and the performance comparison with SSAM, NVCC, PPCG, and Halide is displayed in Figure 5.2. The statement reordering framework (namely reordered in Figure 5.2) achieves better performance than NVCC, PPCG, and Halide.

Furthermore, using the reordering framework that proposed In [132], Rawat et al. demonstrated its effection for performance improvement by relieving register pressure in literature [133]. Using multiple popular compilers (i.e. GCC, Clang/LLVM) and target platforms (i.e. Intel Xeon Phi, and Intel x86 multi-core CPU), the experimental results demonstrate the advance of the proposed algorithms.

It is well-known that the performance of the application can be significantly improved by GPUs and thus, Zhang et al. [171] proposed a register-only algorithm for the 3D 7-point stencil computation.

7.3 Optimizing stencil kernels

Stencils are a class of typical computation patterns in the scientific field and thus, a plethora of work contributes to the optimization of stencil kernels. To reduce the required memory bandwidth by exploring data reuse, spatial blocking and temporal blocking methods are commonly used approaches. Furthermore, several tiling approaches, e.g. overlapped tiling, diamond tiling [15], hybrid tiling [46, 45], are also adopted to improve the computational parallelism.

The 3.5-D blocking algorithm, a combination of 2.5-D spatial blocking and 1-D temporal blocking, was innovated by Nguyen et al. [115] for modern CPUs and GPUs. The key techniques of 3.5-D blocking are overlapped tiling and streaming-based temporal blocking. To leverage the applications of 3.5D blocking, Rawat et al. implemented DSL frameworks, termed as ARTEMIS [130] and StencilGen [135]. Furthermore, to alleviate the pressure due to the exhaustive usage of GPUs resources (i.e. registers and shared memory), in another work [131], Rawat et al. analyzed the required amount of such resources and presented an improved algorithm, namely streaming + registers algorithm (for more details reader can refer to Alg 3 in [131]), in which the thread-private registers are used to store partial summations instead of the temporal value at the specified time step. To author's knowledge, there are still some inefficiencies in such kind of schemes. First, as the fastest on-chip memory, a large number of registers are only employed to store the accumulated partial summation while shared memory is leveraged to perform the arithmetic operations, e.g, multiplication, addition. Second, compared to the operation of direct registers access, loading shared memory to register is implicit yet cost. Finally, the aggregate utilization of memory fence operations degrades the performance of CUDA-based stencil kernels. Hence, we proposed SSAM to achieve to improve the stencil computation on some extent as demonstrated by several stencils in Section 4.5

As one of the most important algorithms for reducing the redundant computation in stencils, hybrid tiling combines both hexagonal tiling and wavefront tiling as in literature [45–47]. In [74], Krishnamoorthy et al. used *overlapped tiling* optimization for the stencil to preserve concurrency in time-tiled computations. This strategy is adopted by SSAM as Figure 4.5 shows. Note that the negative effection of the redundant computation (known as halo layer) is proved to be insignificant by performance model (as in Section 4.4.) and evaluation.

In literature [175], Zumbusch et al. implemented vectorized kernels for high order finite stencils targeting several processor architectures, i.e. AMD/Intel CPUs, Nvidia GPUs. In addition, a collection of optimizing techniques is employed for improving the computational performance, e.g. combination of vectorization, loop unrolling, an interleaved data layout, spatial and temporal loop tiling methodology, and parameter tuning in one to three spatial dimensions, etc.

Rawat et al. [132, 133] proposed a reorder framework to optimize register allocation for both CPUs and GPUs as the performance comparison in earlier sections. StencilGen [134], a state-of-the-art DSL implementing advanced temporal blocking for stencils.

There exist several approaches aiming to simplify the programming of stencils. Code automation tools are used for stencil optimizations, including stencil-specific DSL or EDSLs (Embedded DSLs), e.g. HLSF [39], Pochoir stencil compiler [147], Halide [111], PolyMage [112], PPCG [151]. Note that both halide and PPCG are introduced in Section 2.3. Halide is a DSL and ppcg [151] is general code-to-code transformation framework. Both generate code for GPUs, yet the automated code does not always yield the ideal performance as demonstrated in literature [134] and our evaluation in Section 4.5.

Hagedorn et al. [51] expressed stencils and their optimizations in LIFT, which is a data-parallel, hardware-agnostic intermediate language. By extending two primitives to gather neighboring stencil elements (named as a slide) and define boundary conditions (named as pad), LIFT can optimize complex stencils allowing higher-level DSLs to be defined by these primitives. Comparing the PPCG polyhedral GPU compiler, LIFT achieved better performance in many cases.

7.4 Partial sums method

Partial sums method is a simple yet effective optimization methodology for a wide range of applications. In SSAM, we take advantage of registers to accumulate the partial sums due to its characteristics of high throughput and low latency.

In [3], Basu et al. incorporate partial sums methodology into CHiLL compiler for generating SIMD-optimized code for CPUs. Using four different Jacobi smoothers, i.e. 7pt, 13pt, 27pt, and 125pt, the authors obtained 4× speedup on the smoothers themselves and up to a 3× speedup on the multigrid solver.

To parallelize sequential loops with recurrences, Jiang et al. [65] proposed a novel sampling-and-reconstruction method to speed up their computations in an automated fashion, involving refined runtime-checking rules and resulting in reducing the overhead of reprocessing. In [167], Xia et al. proposed methods to merge partial results in parallel on GPUs and achieve linear scalability for optimizing loops with recurrences.

Parallel scan operators highly rely on the partial sums accumulation as details in Algorithm 4, Algorithm 2, and Algorithm 3. In [94], Lui et al. presented an efficient one-dimensional scan (called LightScan) on CUDA-enabled GPUs. To perform the in-register partial sum accumulation, their algorithm employs shuffle intrinsic to perform the fast intra-warp computation and uses globally coherent L2 cache and the associated parallel thread execution (PTX) assembly instructions to realize lightweight inter-block communication. Evaluated on a single Tesla K40c GPU, the results demonstrate that LightScan outperforms state-of-the-art algorithms and libraries as up to 2.1, 2.4, 1.5 and 1.2 faster than CUDPP, Thrust, ModernGPU and CUB implementations, respectively.

The conventional algorithms for SAT computation use shared memory to accumulate the partial sums, e.g. OpenCV [18], Nvidia NPP. The demonstrated performance in Section 6.5 shows the advance of SSAM-based implementations using registers rather than shared memory.

7.5 Modeling GPU performance

A performance model suitable for runtime estimation of modern GPUs often needs to characterize the complex interactions of the most important hardware and software features, e.g. GPU resources, CUDA driver, compiler. In [33], Dao et al. presented a linera and an ML-based performance estimation model for GPUs with considering GPU caches. By running a wide range of benchmarks with several OpenCL kernels on Nvidia GPUs and AMD GPUs, the linear model outperforms state-of-the-art models and achieves error rate less than 10%. The ML-based model performs extremely well as the error rate less than 5%.

In [48], Gupta et al. presented an algorithm that combines offline data collection and online learning to construct an adaptive runtime performance model for GPUs. Extensive evaluations on a commercial platform by several GPU benchmarks indicate the achievement is

that in average mean absolute errors of 3.1% in frame time and 3.9% in frame time-sensitivity prediction.

In [99], Ma et al. develop an integrated analytical framework by combining two existing models (asymptotic models and calibrated models) for modeling scheduling mechanisms of GPUs and incorporating both the computation complexity and memory complexity. By doing so, it builds a bridge between the two models for performance analysis of applications on many-core GPUs.

In [98], the authors presented a GPU performance model for Deep Learning Applications with In-depth Memory System Traffic Analysis, termed as DeLTA. The evaluation indicates that DeLTA can model the expected performance of a convolution layer on a GPU and can be used to explore the design space for tuning resource provisioning for CNNs.

Employing the collaborating filtering based modeling technique, the authors in [140] extended an analytical model to predict the performance of a variant of applications that are optimized on GPU systems. Using and seven Nvidia GPUs, the evaluation of a set of well-known micro-applications demonstrated the model can predict with 90.6% accuracy on average.

7.6 Summary

The majority of the work above is limited in scope to specific applications (e.g. stencil), and architectures. Moreover, they lack performance models for the redundancy and do not optimize for reuse in register cache. In [91], Li et al. proposed a general model for all generations of GPUs, called Warp Consolidation, however, it is focused on improving the Cooperative-Thread-Array execution on GPUs.

Using a Stateful DataFlow multiGraph (called as SDFG) to optimize HPC applications, Ben-Nun et al. [7] demonstrated a very general IR (Intermediate Representation) for several accelerators, e.g. CPUs, GPUs, FPGAs. SDFG is reported to achieve high performance by considering more characteristics of GPU architectures.

To the best of our knowledge, this work is the first to propose a completely generic and versatile method, driven by a performance model, to use locality-optimized aregister cache and shuffle instruction for directly computing different filter/stencil sizes for both 2D and 3D grids. Last but most importantly, we pave a novel way to optimize algorithms on CUDA-enabled GPUs: mapping the CUDA cores to software systolic arrays. Thereby, we build a bridge between code automation and optimization.

Chapter 8

Discussion & Conclusion

In this chapter, we discuss SSAM-based optimization and its suitable applications as follows. First, we explain how to extract the program dependency (the D in SSAM) and represent it in the polyhedral model that is a computable framework. Second, we present the possibility of generating SSAM-optimized code automatically. Third, we show the novel optimization approach that improves the dependency of SSAM, resulting in better performance as the use of BRLT in SAT computation. Fourth, we insight into Pascal and Volta architecture for explaining why SSAM performs differently on them. Fifth, we leverage SSAM to the popular accelerators, e.g. CPUs, AMD GPUs, etc. Sixth, we analyze applications that are suitable for SSAM. Finally, we conclude.

8.1 Discussion

In this section, firstly we present our insights into Pascal and Volta architecture. Secondly, we discuss the available algorithms that can be mapped in SSAM.

8.1.1 Automation and Extracting Dependency (D) for SSAM

This study proposes a method for executing kernels with regular data-access behavior in a systolic fashion. The study is dedicated to explaining the method in a comprehensive way that covers the systolic representation/mapping, a performance model, detailed implementation, and a broad range of experiments and comparisons. It is important to note that all kernels evaluated in earlier chapters are strictly implemented as Listing 4.1 and Listing 5.1 without extra manual optimization. Taking Figure 4.6 for instance, Listing 4.1, a single template kernel function, is used for all SSAM-based 2D convolution evaluations.

That being said, it is a non-trivial task for users to manually apply SSAM in large codebases made of tens kernels. An ideal case would be to automate SSAM, by using a DSL or code transformation. We argue that code transformation is the more practical approach since users can be reluctant to move to new DSLs. To apply SSAM as an automated code transformation, extracting the dependency and mapping it to SSAM is an important point to consider when generalizing SSAM to different types of algorithms. Automatic code generation by SSAM is future work that is outside the scope of this study, and the nontrivial body of work required to extract dependencies and map them to SSAM for any given kernel.

In this paragraph, we briefly discuss the possibility of representing the dependency graph (namely the D in Equation 3.2) using the polyhedral model [162, 43, 95]. The polyhedral model is a well-researched mathematical framework for performance optimization [10], which often involves nested loops and large numbers of operations, i.e. convolution, matrix multiplication. According to our formulation in Equation 3.2, and motivational examples in earlier sections, such a parametric representation can be analyzed for mapping algorithms and code, i.e. C/C++, to the SSAM-based intermediate representation (IR).

8.1.2 BRLT algorithm in SSAM

This section discusses the BRLT algorithm in SSAM optimization. As introduced in Section 6.3, we innovate the BRLT algorithm that transposes the register matrix of size 32×32 locally. It is very important to note that we take advantage of the register matrix of size 32×32 to cache data and benefit from the transpose operation as that increasing the parallelism and reducing the arithmetic computation. Despite the BRLT operation is overhead in the original algorithm, its negative affection to the overall performance is ignorable as proved by the performance model (as in Section 6.4) and evaluated results (as in Section 6.5). Furthermore, the BRLT algorithm enhances the importance of Dependency (D) (as in Equation 3.2) in SSAM as discussed in Section 4.4.4. It is because the BRLT improves the algorithm dependency that we can use the simple serial scan rather than the parallel warp scan in SAT computation. Except for taking advantage of in-register partial sum computation in SSAM, BRLT provides a special for application optimization. Hence, BRLT can be used in several algorithms that can be mapped in SSAM, e.g. FFT, DCT, Wavelet Transformation, etc.

8.1.3 Insight into Pascal and Volta architecture

In this section, we outline the reasons why SSAM performs differently on Tesla P100 and V100 GPUs for all evaluated applications, e.g. convolution, stencils, and SAT primitive. The main finding is the improvement of L1/L2 cache on V100 GPU as follows.

(i) The L1 cache in Volta is significantly enhanced. (1) According to Nvidia user guide [120], the **capacity** of L1 cache in Volta has increased up to 128KB, which is more than 7× larger than Pascal; (2) Regarding the access **latency** of L1 cache, V100 is about 2.8× faster than the P100 (as reported in [62]: V100 is 28 cycles vs. P100 is 82 cycles). This narrows the gap between applications that are manually optimized by caching data in registers (or shared memory) and those that access global memory directly. Hence, the difference in performance between the SSAM and other implementations become smaller on the V100 GPU.

(ii) The improvement of L2 cache also contributes to the change in performance behavior between P100 and V100 GPUs. According to the experiments in [62], in comparison to P100, the **capacity** of L2 cache in V100 increases by 50% (P100 is 4096KB vs. V100 is 6144KB), the access **latency** improves by up to 10% (P100 is $\leq$234 cycles vs. V100 is $\leq$193 cycles).

(iii) As Jia et al. [64] illustrated, due to register bank conflict in Volta architecture, MAD instruction (accessing 3 registers in the same cycle) results in one clock stall. More specifically, in Volta, registers are divided into two banks. However, other generations (e.g. Pascal, Maxwell, Kepler) have four banks [1, 172]. Finally, SSAM performs as well as Pascal in Maxwell and Kepler architectures.

8.1.4 Applications that are suitable for SSAM

We analyze the applications suitable for SSAM optimization in this section. The algorithms that can benefit from SSAM optimization have characteristics as follows.

- Memory-bound kernel
- Regular memory access

As introduced in Section 3.2, we take advantage of register files to cache data from global memory, resulting in increasing the data reuse and performing efficient in-register computation, namely register cache in SSAM. To meet the requirement of regular memory access for memory-bound kernels, we employ register cache in SSAM with the following considerations. (i) Coalesced memory access is the basic requirement for improving the read/write performance of global memory. As illustrated by Ben-Sasson et al. [8], register cache performs at warp-level and thus, the regular memory access fashion is suitable for the mechanism of register cache since the successive threads in a single warp can access the contiguous global memory. It is clear that such an approach also meets the access requirement

of L1/L2 caches as introduced in Section 2.1.3. (ii) Due to the utilization of register cache, the memory traffic for global memory access can be greatly reduced. The memory-bound kernel can benefit from such a technique easily. Note that such a scenario can be easily proved by The roofline model as we discussed on the memory bandwidth optimization in Section 3.1.1.

Meanwhile, the algorithms that can be implemented by hardware systolic are also suitable for SSAM since SSAM is a virtual yet versatile systolic array. Regarding PEs in hardware systolic array, the operation is as simple as addition and multiply operations. However, SSAM can perform any operations, namely the O in model J (as in Equation 3.2) that are provided by CUDA architecture, e.g. *sqrt*, *exp*, etc. It is worth mentioning that we list some hardware systolic array applications in Section 2.2.2 and those algorithms are available for SSAM-based optimization theoretically.

8.1.5 Wider Warp

SSAM is a warp-centric approach and benefits from the mechanism of warp as follows. (i) A single warp performs in a SIMD fashion. (ii) Regarding the overall performance, the intra-warp synchronization within a warp is very costly, but inter-warp synchronization is very light-weight. However, the WarpSize is limited as 32 due to the CUDA architecture restrictions. In other words, the maximal number of threads in a single warp is 32. SSAM can benefit from the wider warp in many aspects. (i) Better parallelism for the SIMD-liked computation (ii) Smaller halo-layer for the overlapped tiling technique(as in Section 4.3.5) It is noteworthy that the tendency of the bits in a part of Intel products (as Table 8.1 shown) can enhance the benefits of wider warp in some extent.

Table 8.1 Bits of SIMD in Intel products.

Year	Product	SIMD	bits
1999	Pentium III	SSE	64
2000	Pentium 4	SSE2	128
2003	Pentium M	SSE3	128
2007	Intel Core 2	SSE4.1	128
2008	Intel Core i7	SSE4.2	128
2011	Intel Core i7	AVX	256
2013	Intel Core i7	AVX2	256
2016	Xeon Phi	AVX-512	512

8.2 SSAM for other accelerators

SSAM is not limited to Nvidia GPUs and can be easily extended to other popular accelerators, e.g. Intel CPUs, Arm-based CPUs, AMD GPUs, FPGAs, etc.

8.2.1 AMD GPUs

In this section, we focus on discussing leveraging SSAM to AMD GPUs. Note that on AMD GPUs, Graphics Core Next (GCN) [161] abstracts both of the micro-architecture and its instruction set. GCN was developed by AMD for its GPUs as the successor of the TeraScale micro-architecture and instruction set.

On Nvidia GPUs, the basic execution unit is called Warp; On AMD GPUs, such a basic execution unit is called Wavefront as introduced in literature [136]. For simplification, both Warp and Wavefront are SIMD vectors. On Nvidia GPUs, all threads in a Warp perform in a locked step. Unlike the low-level implementations of CUDA Warp, in a single Wavefront, there are 64 parallel work items (or thread), called lanes. However, the actual hardware is implemented as 16-wide SIMD and thus, a Wavefront decomposes into groups of 16 lanes (termed as Wavefront rows), which are required four consecutive cycles for execution. In Listing 8.1, a 64-points scan operation is implemented by GCN instructions using a single Wavefront. More specifically, we only take 7 steps to achieve the prefix-sum result without using scratchpad memory. Note that the more details on GCN instructions can be found in [136].

Listing 8.1 Parallel scan by a single Wavefront using GCN at single precision.

```
1  v_add_f32 vec1, vec0, vec0 row_shr:1 bound_ctrl:0 // step 1
2  v_add_f32 vec1, vec0, vec1 row_shr:2 bound_ctrl:0 // step 2
3  v_add_f32 vec1, vec0, vec1 row_shr:3 bound_ctrl:0 // step 3
4  v_nop // avoid a data hazard
5  v_nop
6  v_add_f32 vec1, vec1, vec1 row_shr:4 bank_mask:0xe // step 4
7  v_nop // avoid a data hazard
8  v_nop
9  v_add_f32 vec1, vec1, vec1 row_shr:8 bank_mask:0xc // step 5
10 v_nop // avoid a data hazard
11 v_nop
12 v_add_f32 vec1, vec1, vec1 row_bcast:15 row_mask:0xa // step 6
13 v_nop // Add two independent steps to avoid a data hazard
14 v_nop
15 v_add_f32 vec1, vec1, vec1 row_bcast:31 row_mask:0xc // step 7
```

8.2.2 CPUs with vectorized instructions

Modern CPUs (e.g. Intel CPUs, AMD CPUs, ARM-based CPUs) increasingly rely on parallelism to improve the computational performance. Task parallelism is one of the well-known approaches and is often supported at the hardware level, i.e. multi-core, hyperthreading, etc. However, another popular approach is at the instruction level, namely Single Instruction Multiple Data (SIMD). SIMD technique indicates that the CPUs can execute multiple same instructions simultaneously within a single thread. In other words, CPUs are capable of deploying a set of vector operations by SIMD intrinsics, e.g. operating on 4 or 8 inputs and yielding 4 or 8 output in a single cycle.

As introduced earlier, CUDA Warp and GCN Wavefront perform operations in a SIMD fashion. Hence, SSAM can be easily extended to SIMD-supported CPUs. As Listing 8.2 shown, we display the implementation of a BRLT-based parallel scan for a matrix of size 4×4 using SSE intrinsic (BRLT can be found in Chapter 6). The shuffle intrinsic is used to transpose the register matrix as in function *transpose4x4* (line 2∼12), the add operation intrinsic is employed to accumulate the partial sums as in function *ScanRow4x4* (line 15∼22). It is noteworthy that BRLT can be used to transpose the register matrix with larger sizes, e.g. 8×8, 16×16, etc.

Listing 8.2 Parallel scan row for a matrix of size 4×4 at single precision by SSE intrinsic.

```
1  //transpose a register matrix of size 4x4
2  void transpose4x4(__m128& Row0, __m128& Row1, __m128& Row2, __m128& Row3)
3  {
4     __m128 Var0 = _mm_shuffle_ps(Row0, Row1, 0x44);
5     __m128 Var2 = _mm_shuffle_ps(Row0, Row1, 0xEE);
6     __m128 Var1 = _mm_shuffle_ps(Row2, Row3, 0x44);
7     __m128 Var3 = _mm_shuffle_ps(Row2, Row3, 0xEE);
8     Row0 = _mm_shuffle_ps(Var0, Var1, 0x88);
9     Row1 = _mm_shuffle_ps(Var0, Var1, 0xDD);
10    Row2 = _mm_shuffle_ps(Var2, Var3, 0x88);
11    Row3 = _mm_shuffle_ps(Var2, Var3, 0xDD);
12 }
13
14 //parallel scan row for a matrix of size 4x4
15 void ScanRow4x4(__m128& Row0, __m128& Row1, __m128& Row2, __m128& Row3)
16 {
17    transpose4x4(Row0, Row1, Row2, Row3);   //transpose
18    Row1 = _mm_add_ps(Row0, Row1);          //Row1 = Row0 + Row1;
19    Row2 = _mm_add_ps(Row1, Row2);          //Row2 = Row1 + Row3;
20    Row3 = _mm_add_ps(Row3, Row2);          //Row3 = Row2 + Row3;
21 }
```

8.3 Conclusion

In this study, we present a virtual systolic array that is built on the top of CUDA architecture and explains how to map a set of algorithms in SSAM by three illustrative examples, i.e. convolution, stencils, and SAT primitive. To explain the performance advance of the SSAM-based algorithms, we build performance models to insight the internal mechanism of SSAM-optimized applications. Furthermore, we demonstrate the outstanding performance of SSAM-based algorithms using the latest Nvidia Tesla P100 and V100 GPUs in comparison with state-of-the-art libraries, e.g. NPP, OpenCV, Arrayfire.

GPU accelerator In the past a few years, GPU becomes a more and more important many-core accelerator in a variant of fields, i.e. scientific computation, machine learning. Taking the TOP500 [148] list in June 2019 for example, both the first and second powerful supercomputers use Nvidia Tesla V100 GPUs, i.e. Summit and Sierra supercomputers. As one of the most popular accelerators, GPU accelerates a large number of scientific and engineering applications. However, it is not easy to obtain the peak performance for applications running on GPUs. It is because GPUs have a complex execution and memory hierarchy. Unlike the latency-oriented CPUs, GPUs are throughput-oriented processor and thus, it is fundamentally crucial that launching sufficient threads to meet the requirement of concurrency in all physical cores. In the perspective of execution hierarchy, a large number of CUDA blocks and grids are needed to be allocated. Also, there are a few different types of memories and caches, e.g. global memory, shared memory, constant memory, register memory, L1 and L2 cache, etc. Each type of memory or cache has its characteristics, e.g. latency, throughput, capacity, supported access pattern, and read/write restrictions. Hence, it essential to take advantage of this memory hierarchy to optimize the applications on GPUs.

Systolic array A systolic array is an old yet revived parallel computing architecture. Recently, it is commonly adopted as a co-processor to speed up the domain-specified Deep Learning workload, e.g. Google TPU, Nvidia Tensor Core. We list some advantages of a systolic array as introduced in Section 2.2. Using a set of PEs to compose a two-dimensional network, the systolic array architecture is very simple and regular; A systolic array is capable of achieving extremely high Flop/s with the limited power consumption.

Inspired by the mechanism of the systolic array architecture, we proposed a novel execution model (namely SSAM) on CUDA-enabled GPUs. The SSAM can be seen as a virtual systolic array building on the top of CUDA. It is because SSAM mimics the processing of a systolic array. In other words, we simulate the systolic array using a software implementation. More specifically, we use GPU registers to mimic the processing of PEs,

take advantage of shuffle intrinsic to perform the partial sums communication between the virtual PEs. Since SSAM benefits from characteristics of registers, namely high throughput and low latency, SSAM-optimized applications can achieve very competitive performance as demonstrated in the earlier chapters. Also, as one of the important challenges, the coalesced access for global memory is absorbed in SSAM, namely the X and Y in the formulated Equation 3.2.

8.3.1 SSAM : A virtual systolic array

There is a strong motivation to build a model to formulate the application optimization technique that uses register cache and shuffle intrinsic. To the author's knowledge, all prior researches analyzed and optimized the specified problem on a case-by-case basis. e.g. Eli Ben-Sasson et al. [8] optimized the binary finite field multiplication algorithm, Lai and Seznec [85] suggested a method named register blocking to explore the potential peak performance of SGEMM. Hence, we leverage these optimization techniques by the formulated SSAM model. As demonstrated in the earlier chapters, all examples (i.e. convolution, stencil, SAT) are implemented in a simple and plain style. Furthermore, the register matrix of size 32×32 can be transposed by the *BRLT* algorithm and thus it results in the improvement of dependency for the arithmetic pipeline. At the ignorable cost, the BRLT algorithm can improve the dependency graph which is a key competency in SSAM (as in Equation 3.2).

In this study, we focus on the memory-bound kernels with regular memory access due to memory-bound is a prevalent computing pattern for most of the application on GPUs (as introduced in Section 2.4). SSAM is a versatile and general model for application optimization. It is worth mentioning that the compute-bound problem is also applicable to SSAM and the problems with irregular access pattern is still a challenging work for SSAM as will be our future work. We present details on mapping algorithms in SSAM, i.e. 2D convolution, 2D/3D stencil, and SAT primitive. Note that all of these examples are typical memory-bound kernels with regular memory access. In comparison with the state-of-the-art implementations, we demonstrate the outstanding performance of SSAM-based applications. To justify the validity of SSAM for application optimization, we build a performance model to analyze the mechanism of all detailed parts of the implementations. Using a collection of micro-benchmarks, the latency of arithmetic computation and shared memory access are measured and then applied in the analytical performance model.

Convolution 2D convolution is fundamental in many image processing and Deep Learning applications and a plethora of work contribute to the optimization of 2D convolution. In comparison with the highly optimized libraries, SSAM-based 2D convolution achieves

the best performance in many aspects, e.g. execution time and scalability. Using 2D convolution as a detailed example to illustrate how to map algorithms in SSAM, we list some important techniques, e.g. coalesced access to global memory, partial sums accumulation, and communication, in-register computation, etc.

Stencils The stencil is an important computing pattern and commonly employed scientific applications, e.g. weather, wave, and fluid simulations. Researches have been working for a long time to improve the performance of stencil computations. To improve the parallelism, several tiling algorithms are applied, e.g. overlapped tiling, diamond tiling, hexagon tiling, etc. Considering the characteristics of the GPU architecture, e.g. SIMT execution model and coalesced access to global memory, we adopted overlapped tiling in all of the SSAM-optimized applications. Note that such a strategy is also applied in the highly optimized libraries, e.g. StencilGen. Spatial blocking and temporal blocking are widely used techniques to improve the data locality and data reuse for stencil computations. Using a serial implementation and evaluation, we demonstrate such techniques are applicable in SSAM and are capable of optimizing stencil code for high performance.

SAT primitive Similar to FFT computation, SAT, an important primitive, is critical to the performance of a variant of applications, e.g. real-time face detection, image recognition. We map the SAT to SSAM by a collection method and innovate an efficient algorithm to transpose the small register matrix of size 32×32. We proposed three SSAM-based algorithms to improve the SAT computation, i.e. *ScanRowColumn*, *BRLT-ScanRow*, and *ScanRow-BRLT*. The proposed BRLT method is general and can be applied to optimize many orthogonal transformation algorithms, such as Fast Fourier Transform (FFT), Wavelet Transform, Discrete Cosine Transform (DCT), etc.

8.3.2 Register pressure

Since SSAM highly relies on the GPU register cache, the performance of the SSAM-optimized application may be affected by versions of CUDA drivers and NVCC versions. The major reason is that we can not control the physical register allocation or map the register variables to the physical ones. What makes it more difficult is that the NVCC compiler is closed-source and the low-level assembler for GPUs is inaccessible. Additionally, the architecture difference of GPUs also affects the performance of the SSAM-optimized applications. The improved L1 and L2 caches may narrow the gap between SSAM and other kinds of implementations, e.g. directly access global memory.

8.3.3 Code automation

The simplicity and generality of SSAM denote the availability of SSAM for code automation. Using the polyhedral dependence analysis, several applications can be mapped to SSAM. To be emphasized that the application is required to access memory in a regular pattern. In this study, we only present three typical examples of how to map them in SSAM and generating SSAM-optimized code for GPUs will be our future work. The evaluation shows that SSAM-optimized code outperforms the state-of-the-art tools, e.g. PPCG, Halide. This is one of the strongest motivations for code automation in SSAM. The DSL-based StencilGen can generate stencil code while achieving extremely high performance. However, it is only specified in stencil computation while SSAM is more general and versatile.

Chapter 9

Future Work

This chapter discusses the future work for SSAM. It is noteworthy that **SSAM is not the replacement, however, the enhancement of the shared memory-based programming model, which is the recommended by Nvidia and is a widely used model in CUDA-optimized applications.** To leverage SSAM for more and more applications, there are still several works in the future. In the coming years, research on SSAM code automation and application optimization will potentially play an important role in tackling various challenges of both scientific and HPC fields. We will leverage SSAM as a fundamental technique for tuning many kinds of applications while overcoming its limitations, e.g. finding a solution for alleviating register pressure. In addition, we will extend SSAM to other popular accelerators, e.g. FPGA, ARM-based CPUs.

9.1 SSAM code automation

This study focus on improving the performance of the memory-bound kernels with regular memory access by mapping such algorithms to a virtual systolic array, namely SSAM. In the future, we will consider the compute-bound problems (e.g. Matrix Multiplication [93]), even the more challenging problems that memory-bound kernels with irregular memory access (e.g. Sparse matrix-vector multiplication [6], Sparse matrix-matrix multiplication [31], triangular matrices [57].). We will also provide a tool that can generate SSAM-based codes using polyhedral analysis.

9.2 Warp specification for hiding memory latency

We will use warp specification [4] technique for hiding memory latency. Regarding X and Y in model J (as in Equation 3.2), their operations can be further optimized. Tuning the performance of GPU kernels is primarily about managing the restricted resources, e.g. register and shared memory, instruction issue, and memory bandwidth. Warp specialization is an optimizing method allowing subsets of threads within a CUDA block to execute for a particular purpose, which results in more efficient consumption of the limited resources. Several techniques within Warp Specification are listed in literature [4]. Using the fine-grained synchronization and buffering techniques, the compute Warps and DMA Warps execute in parallel, which results in hiding the memory access latency within the computations.

9.3 Alleviate register pressure

As introduced in Section 2.1.3, registers are the fastest on-chip memory and constrained with a limited capacity. Due to a large amount of use of register files, alleviating register pressure is considered a promising way to improve the performance of applications on GPUs. Rawat et al. proposed a method, called Associative Instruction Reordering, to achieve this goal by optimizing the register allocation at the compiler level. We plan to incorporate this technique into our SSAM model.

9.4 A virtual systolic array on CPUs

The proposed SSAM model can be extended to vectorized CPUs. Since a single Warp performs in a SIMD fashion, such a virtual systolic array is capable of emulated on CPUs by the SIMD instructions. However, the architecture of GPUs is completely different from the CPUs, it is required to improve the SSAM model to meet the constraint of CPUs.

9.5 Automate systolic array on FPGA

FPGA [54, 122], a reconfigurable logic arrays, allows for synthesizing a variant of virtual circuits. Authors in [155] leverage systolic array architecture on FPGAs for Convolutional Neural Networks (CNNs) workload and achieve high clock frequency under high resource utilization. Cong et al. [27] proposed an automatic systolic array generation (called as PolySA) that can generate high-performance systolic array architectures on FPGAs using high-level synthesis. Using matrix multiply and CNN as an illustrative example, PolySA

demonstrates the possibility of generating systolic array architectures on FPGAs. Thereby, we can challenge to generate a systolic array on FPGAs via the SSAM model and thus, leverage SSAM-based optimization on FPGA applications.

9.6 Performance modeling

With the exponential growth of SMs in the latest GPUs, how to analyze and optimizing its performance becomes very challenging. In the future, we will extend and apply our analytical model to more general applications. Thereby, we can guide the programmers to identify the bottleneck of their applications. Note that not only the SSAM-based applications but also any other general ones. More specifically, we will model more details of CUDA architecture, e.g. L1 and L2 caches, compute-units.

9.7 Multi-GPUs optimization

To tackle increasingly complex problems, we can utilize multiple GPUs to solve such problems. Facilitated by the SSAM-optimized CUDA kernels, we can achieve higher performance in computations. However, the communications between GPUs may become a bottleneck of such kind of implementations and thus, it becomes a case-to-case analysis.

References

[1] Anderson, M. J., Sheffield, D., and Keutzer, K. (2012). A predictive model for solving small linear algebra problems in gpu registers. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 2–13.

[2] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. (2009). A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67.

[3] Basu, P., Hall, M., Williams, S., Straalen, B. V., Oliker, L., and Colella, P. (2015). Compiler-directed transformation for higher-order stencils. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 313–323.

[4] Bauer, M., Cook, H., and Khailany, B. (2011). Cudadma: optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, page 12. ACM.

[5] Bay, H., Tuytelaars, T., and Van Gool, L. (2006). Surf: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer.

[6] Bell, N. and Garland, M. (2009). Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*, page 18. ACM.

[7] Ben-Nun, T., Licht, J. d. F., Ziogas, A. N., Schneider, T., and Hoefler, T. (2019). Stateful dataflow multigraphs: A data-centric model for high-performance parallel programs. *arXiv preprint arXiv:1902.10345*.

[8] Ben-Sasson, E., Hamilis, M., Silberstein, M., and Tromer, E. (2016a). Fast multiplication in binary fields on gpus via register cache. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 35:1–35:12, New York, NY, USA. ACM.

[9] Ben-Sasson, E., Hamilis, M., Silberstein, M., and Tromer, E. (2016b). Fast multiplication in binary fields on gpus via register cache. In *Proceedings of the 2016 International Conference on Supercomputing*, page 35. ACM.

[10] Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A., and Bastoul, C. (2010). The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*, pages 283–303. Springer.

[11] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, pages 1–7.

[12] Bhatnagar, H. (2002). *Advanced ASIC chip synthesis*. Springer.

[13] Bilgic, B., Horn, B. K., and Masaki, I. (2010). Efficient integral image computation on the gpu. In *Intelligent vehicles symposium (IV), 2010 IEEE*, pages 528–533. IEEE.

[14] Blelloch, G. E. (1990). Prefix sums and their applications.

[15] Bondhugula, U., Bandishti, V., and Pananilath, I. (2017). Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1285–1298.

[16] Borkar, S. (1999). Design challenges of technology scaling. *IEEE micro*, 19(4):23–29.

[17] Bradley, D. and Roth, G. (2007). Adaptive thresholding using the integral image. *Journal of graphics tools*, 12(2):13–21.

[18] Bradski, G. and Kaehler, A. (2000). Opencv. *Dr. Dobb's journal of software tools*, 3.

[19] Brent, R. P. and Kung, H.-T. (1982). A regular layout for parallel adders. *IEEE transactions on Computers*, (3):260–264.

[20] Brent, R. P. and Kung, H.-T. (1984). Systolic vlsi arrays for polynomial gcd computation. *IEEE Transactions on Computers*, 100(8):731–736.

[21] Chellapilla, K., Puri, S., and Simard, P. (2006). High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft.

[22] Chen, P., Wahib, M., Takizawa, S., Takano, R., and Matsuoka, S. (2018). Efficient algorithms for the summed area tables primitive on gpus. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 482–493. IEEE.

[23] Cheng, C. and Parhi, K. K. (2005). A novel systolic array structure for dct. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 52:366–369.

[24] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. (2014). cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*.

[25] Chrysos, G. (2014). Intel® xeon phi™ coprocessor-the architecture. *Intel Whitepaper*, 176.

[26] Cole, R. and Vishkin, U. (1989). Faster optimal parallel prefix sums and list ranking. *Information and computation*, 81(3):334–352.

[27] Cong, J. and Wang, J. (2018). Polysa: polyhedral-based systolic array auto-compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE.

[28] Cook, S. (2013). *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[29] Crow, F. C. (1984). Summed-area tables for texture mapping. In *ACM SIGGRAPH computer graphics*, volume 18, pages 207–212. ACM.

[30] CUDA, N. (2019). Cuda toolkit documentation. *NVIDIA Developer Zone. http://docs.nvidia.com/cuda/index.html*.

[31] Dalton, S., Olson, L., and Bell, N. (2015). Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)*, 41(4):25.

[32] Dang, Q., Yan, S., and Wu, R. (2014). A fast integral image generation algorithm on gpus. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pages 624–631. IEEE.

[33] Dao, T. T., Kim, J., Seo, S., Egger, B., and Lee, J. (2014). A performance model for gpus with caches. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):1800–1813.

[34] Davidson, A. and Owens, J. D. (2011). Register packing for cyclic reduction: A case study. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 4:1–4:6, New York, NY, USA. ACM.

[35] Deest, G., Estibals, N., Yuki, T., Derrien, S., and Rajopadhye, S. (2016). Towards scalable and efficient fpga stencil accelerators.

[36] Diéguez, A. P., Amor, M., and Doallo, R. (2014). Efficient scan operator methods on a gpu. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, pages 190–197. IEEE.

[37] Dooply, A. E. (2005). System and method for systolic array sorting of information segments. US Patent 6,879,596.

[38] Dreger, D. S., Huang, M.-H., Rodgers, A., Taira, T., and Wooddell, K. (2015). Kinematic finite-source model for the 24 august 2014 south napa, california, earthquake from joint inversion of seismic, gps, and insar data. *Seismological Research Letters*, 86(2A):327–334.

[39] Dütsch, F., Djelassi, K., Haidl, M., and Gorlatch, S. (2014). Hlsf: A high-level; c++-based framework for stencil computations on accelerators. In *WOSC '14*.

[40] Ehsan, S., Clark, A. F., McDonald-Maier, K. D., et al. (2015). Integral images: efficient algorithms for their computation and storage in resource-constrained embedded vision systems. *Sensors*, 15(7):16804–16830.

[41] Enfedaque, P., Auli-Llinas, F., and Moure, J. C. (2015). Implementation of the dwt in a gpu through a register-based strategy. *IEEE Transactions on Parallel and Distributed Systems*, 26(12):3394–3406.

[42] Fernandes, K. and Cardoso, J. S. (2017). Deep local binary patterns. *arXiv preprint arXiv:1711.06597*.

[43] Griebl, M., Lengauer, C., and Wetzel, S. (1998). Code generation in the polytope model. In *In IEEE PACT*, pages 106–111. IEEE Computer Society Press.

[44] Gross, T. and O'Hallaron, D. R. (1998). *iWarp: anatomy of a parallel computing system*. Mit Press.

[45] Grosser, T., Cohen, A., Holewinski, J., Sadayappan, P., and Verdoolaege, S. (2014a). Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 66. ACM.

[46] Grosser, T., Verdoolaege, S., Cohen, A., and Sadayappan, P. (2013). *The Promises of Hybrid Hexagonal/Classical Tiling for GPU*. PhD thesis, INRIA.

[47] Grosser, T., Verdoolaege, S., Cohen, A., and Sadayappan, P. (2014b). The relation between diamond tiling and hexagonal tiling. *Parallel Processing Letters*, 24(03):1441002.

[48] Gupta, U., Campbell, J., Ogras, U. Y., Ayoub, R., Kishinevsky, M., Paterna, F., and Gumussoy, S. (2016). Adaptive performance prediction for integrated gpus. In *Proceedings of the 35th International Conference on Computer-Aided Design*, page 61. ACM.

[49] Gustafson, J. L. (1988). Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533.

[50] Ha, S.-W. and Han, T.-D. (2013). A scalable work-efficient and depth-optimal parallel scan for the gpgpu environment. *IEEE Transactions on Parallel and Distributed Systems*, 24(12):2324–2333.

[51] Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., and Dubach, C. (2018). High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 100–112. ACM.

[52] Han, B., Yang, C., Duraiswami, R., and Davis, L. (2005). Bayesian filtering and integral image for visual tracking. In *Workshop on Image Analysis for Multimedia Interactive Services*. Citeseer.

[53] Harris, M., Sengupta, S., and Owens, J. D. (2007). Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876.

[54] Hauck, S. and DeHon, A. (2010). *Reconfigurable computing: the theory and practice of FPGA-based computation*, volume 1. Elsevier.

[55] Hensley, J., Scheuermann, T., Coombe, G., Singh, M., and Lastra, A. (2005). Fast summed-area table generation and its applications. In *Computer Graphics Forum*, volume 24, pages 547–555. Wiley Online Library.

[56] Hensley, J., Scheuermann, T., Singh, M., and Lastra, A. (2007). Fast hdr image-based lighting using summed-area tables. In *Symposium on Interactive 3D Graphics and Games (Poster)*. Citeseer.

[57] Higham, N. J. (1987). A survey of condition number estimation for triangular matrices. *Siam Review*, 29(4):575–596.

[58] Hill, M. D. and Marty, M. R. (2008). Amdahl's law in the multicore era. *Computer*, 41(7):33–38.

[59] Hou, K., Liu, W., Wang, H., and Feng, W.-c. (2017). Fast segmented sort on gpus. In *Proceedings of the International Conference on Supercomputing*, page 12. ACM.

[60] Iyengar, G. and Panchanathan, S. (1995). Systolic array architecture for gabor decomposition. *IEEE transactions on circuits and systems for video technology*, 5(4):355–359.

[61] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM.

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

[63] Jia, Z., Maggioni, M., Staiger, B., and Scarpazza, D. P. (2018). Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *ArXiv e-prints*.

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

[65] Jiang, P., Chen, L., and Agrawal, G. (2018). Revealing parallel scans and reductions in recurrences through function reconstruction. pages 1–13.

[66] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. (2017). In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE.

[67] Kågström, B., Ling, P., and Van Loan, C. (1998). Gemm-based level 3 blas: High-performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software (TOMS)*, 24(3):268–302.

[68] Kasagi, A., Tabaru, T., and Tamura, H. (2017). Fast algorithm using summed area tables with unified layer performing convolution and average pooling. In *Machine Learning for Signal Processing (MLSP), 2017 IEEE 27th International Workshop on*, pages 1–6. IEEE.

[69] Kerr, A., Diamos, G., and Yalamanchili, S. (2010). Modeling gpu-cpu workloads and systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 31–42. ACM.

[70] Kilts, S. (2007). *Advanced FPGA design: architecture, implementation, and optimization*. John Wiley & Sons.

[71] Kloeden, P. E. and Pearson, R. (1977). The numerical solution of stochastic differential equations. *The ANZIAM Journal*, 20(1):8–12.

[72] Kogge, P. M. and Stone, H. S. (1973). A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE transactions on computers*, 100(8):786–793.

[73] Kolba, D. and Parks, T. (1977). A prime factor fft algorithm using high-speed convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 25(4):281–294.

[74] Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A., and Sadayappan, P. (2007). Effective automatic parallelization of stencil computations. *SIGPLAN Not.*, 42(6):235–244.

[75] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

[76] Kumar, V. P. and Tsai, Y.-C. (1991). On synthesizing optimal family of linear systolic arrays for matrix multiplication. *IEEE Transactions on Computers*, 40(6):770–774.

[77] Kung, H. (1979). Let's design algorithms for vlsi systems.

[78] Kung, H. and Picard, R. (1981). Hardware pipelines for multi-dimensional convolution and resampling. In *Proceedings of the 1981 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, pages 273–278.

[79] Kung, H.-T. (1982). Why systolic architectures? *IEEE computer*, 15(1):37–46.

[80] Kung, H. T. and Leiserson, C. E. (1980). Algorithms for vlsi processor arrays.

[81] Kung, S. (1985). Vlsi array processors. *IEEE ASSP Magazine*, 2(3):4–22.

[82] Kung, S. Y. (1988). Vlsi array processors. *Englewood Cliffs, NJ, Prentice Hall, 1988, 685 p. Research supported by the Semiconductor Research Corp., SDIO, NSF, and US Navy.*

[83] Kurzak, J., Luszczek, P., Gates, M., Yamazaki, I., and Dongarra, J. (2013). Virtual systolic array for qr decomposition. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 251–260. IEEE.

[84] Ladner, R. E. and Fischer, M. J. (1980). Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4):831–838.

[85] Lai, J. and Seznec, A. (2013). Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE Computer Society.

[86] Lam, M. S. (2012). *A systolic array optimizing compiler*, volume 64. Springer Science & Business Media.

[87] Lang, H.-W., Schimmler, M., Schmeck, H., and Schroder, H. (1985). Systolic sorting on a mesh-connected network. *IEEE Transactions on Computers*, (7):652–658.

[88] Lavin, A. and Gray, S. (2016). Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021.

[89] Lee, M. H., Park, J. O., and Yasuda, Y. (1990). Simple systolic arrays for discrete cosine transform. *Multidimensional Systems and Signal Processing*, 1(4):389–398.

[90] Lewis, J. P. (1995). Fast template matching. In *Vision interface*, volume 95, pages 15–19.

[91] Li, A., Liu, W., Wang, L., Barker, K., and Song, S. L. (2018). Warp-consolidation: A novel execution model for gpus. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, pages 53–64, New York, NY, USA. ACM.

[92] Li, G.-J. et al. (1985). The design of optimal systolic arrays. *IEEE Transactions on Computers*, 100(1):66–77.

[93] Li, Y., Dongarra, J., and Tomov, S. (2009). A note on auto-tuning gemm for gpus. In *International Conference on Computational Science*, pages 884–892. Springer.

[94] Liu, Y. and Aluru, S. (2016). Lightscan: Faster scan primitive on cuda compatible manycore processors. *arXiv preprint arXiv:1604.04815*.

[95] Loechner, V. (1999). Polylib: A library for manipulating parameterized polyhedra.

[96] Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110.

[97] Lu, P. J., Oki, H., Frey, C. A., Chamitoff, G. E., Chiao, L., Fincke, E. M., Foale, C. M., Magnus, S. H., McArthur, W. S., Tani, D. M., et al. (2010). Orders-of-magnitude performance increases in gpu-accelerated correlation of images from the international space station. *journal of real-time image processing*, 5(3):179–193.

[98] Lym, S., Lee, D., O'Connor, M., Chatterjee, N., and Erez, M. (2019). Delta: Gpu performance model for deep learning applications with in-depth memory system traffic analysis. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 293–303. IEEE.

[99] Ma, L., Chamberlain, R. D., and Agrawal, K. (2014). Performance modeling for highly-threaded many-core gpus. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 84–91. IEEE.

[100] Malas, T. M., Hager, G., Ltaief, H., Stengel, H., Wellein, G., and Keyes, D. E. (2015). Multicore-optimized wavefront diamond blocking for optimizing stencil updates. *SIAM J. Scientific Computing*, 37(4).

[101] Maruyama, N. and Aoki, T. (2014). Optimizing Stencil Computations for NVIDIA Kepler GPUs. In Größlinger, A. and Köstler, H., editors, *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*, pages 89–95, Vienna, Austria.

[102] Mattes, L. and Kofuji, S. (2010a). Overcoming the gpu memory limitation on fdtd through the use of overlapping subgrids. In *2010 International Conference on Microwave and Millimeter Wave Technology*, pages 1536–1539. IEEE.

[103] Mattes, L. and Kofuji, S. (2010b). The use of overlapping subgrids to accelerate the fdtd on gpu devices. In *2010 IEEE Radar Conference*, pages 807–810. IEEE.

[104] Mei, X., Zhao, K., Liu, C., and Chu, X. (2014). Benchmarking the memory hierarchy of modern gpus. In *IFIP International Conference on Network and Parallel Computing*, pages 144–156. Springer.

[105] Messom, C. and Barczak, A. (2008). Stream processing of geometric and central moments using high precision summed area tables. In *International Conference on Neural Information Processing*, pages 1095–1102. Springer.

[106] Micikevicius, P. (2009a). 3D Finite Difference Computation on GPUs Using CUDA. In *Proceedings of 2ND Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84.

[107] Micikevicius, P. (2009b). 3d finite difference computation on gpus using cuda. In *Proceedings of 2nd workshop on general purpose processing on graphics processing units*, pages 79–84. ACM.

[108] Milne, W. E. and Milne, W. (1953). *Numerical solution of differential equations*, volume 19. Wiley New York.

[109] Moldovan, D. I. (1987). Advis: A software package for the design of systolic arrays. *IEEE transactions on computer-aided design of integrated circuits and systems*, 6(1):33–40.

[110] Moreira, G., Feijo, B., Lopes, H., and Feitosa, R. Q. (2013). Real-time object tracking in high-definition video using frame segmentation and background integral images. In *Graphics, Patterns and Images (SIBGRAPI), 2013 26th SIBGRAPI-Conference on*, pages 75–82. IEEE.

[111] Mullapudi, R. T., Adams, A., Sharlet, D., Ragan-Kelley, J., and Fatahalian, K. (2016). Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):83.

[112] Mullapudi, R. T., Vasista, V., and Bondhugula, U. (2015). Polymage: Automatic optimization for image processing pipelines. In *ACM SIGPLAN Notices*, volume 50, pages 429–443. ACM.

[113] Nehab, D., Maximo, A., Lima, R. S., and Hoppe, H. (2011). Gpu-efficient recursive filtering and summed-area tables. In *ACM Transactions on Graphics (TOG)*, volume 30, page 176. ACM.

[114] Nguyen, A., Satish, N., Chhugani, J., Kim, C., and Dubey, P. (2010a). 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13.

[115] Nguyen, A., Satish, N., Chhugani, J., Kim, C., and Dubey, P. (2010b). 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13. IEEE Computer Society.

[116] Niamat, M., Molyet, R., Jamali, M., Cios, K., and Raftopoulos, D. (1989). Applications of systolic array architectures to motion analysis studies. In *Conference Proceeding IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 60–63. IEEE.

[117] Nurdin, D. S., Isa, M. N. M., Ismail, R. C., and Ahmad, M. I. (2018). High performance systolic array core architecture design for dna sequencer. In *MATEC Web of Conferences*, volume 150, page 06009. EDP Sciences.

[118] Nvidia (2017a). Cuda c programming guide.

[119] Nvidia (2017b). Nvidia tesla v100 gpu architecture. *Technical whitepaper. https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf*.

[120] Nvidia (2019). Tuning cuda applications for volta. *https://docs.nvidia.com/cuda/volta-tuning-guide/index.html*.

[121] Nvidia, C. (2018). Cuda c programming guide.

[122] Omondi, A. R. and Rajapakse, J. C. (2006). *FPGA implementations of neural networks*, volume 365. Springer.

[123] Oro, D., Fernández, C., Saeta, J. R., Martorell, X., and Hernando, J. (2011). Real-time gpu-based face detection in hd video sequences. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 530–537. IEEE.

[124] Parhi, K. K. and Nishitani, T. (1993). Vlsi architectures for discrete wavelet transforms. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(2):191–202.

[125] Peterson, C., Sutton, J., and Wiley, P. (1991). iwarp: a 100-mops, liw microprocessor for multicomputers. *IEEE Micro*, 11(3):26–29.

[126] Petkov, N. (1992). *Systolic Parallel Processing*. Elsevier Science Inc., New York, NY, USA.

[127] Poostchi, M., Palaniappan, K., Li, D., Becchi, M., Bunyak, F., and Seetharaman, G. (2017). Fast integral histogram computations on gpu for real-time video analytics. *arXiv preprint arXiv:1711.01919*.

[128] Puchała, D., Stokfiszewski, K., Yatsymirskyy, M., and Szczepaniak, B. (2015). Effectiveness of fast fourier transform implementations on gpu and cpu. In *2015 16th International Conference on Computational Problems of Electrical Engineering (CPEE)*, pages 162–164. IEEE.

[129] Quinton, P. (1983). *The systematic design of systolic arrays*. PhD thesis, INRIA.

[130] Rawat, P. S. (2018). *Optimization of Stencil Computations on GPUs*. PhD thesis, The Ohio State University.

[131] Rawat, P. S., Hong, C., Ravishankar, M., Grover, V., Pouchet, L.-N., and Sadayappan, P. (2016). 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*, pages 92–102. ACM.

[132] Rawat, P. S., Rastello, F., Sukumaran-Rajam, A., Pouchet, L.-N., Rountev, A., and Sadayappan, P. (2018a). Register optimizations for stencils on gpus. *SIGPLAN Not.*, 53(1):168–182.

[133] Rawat, P. S., Sukumaran-Rajam, A., Rountev, A., Rastello, F., Pouchet, L.-N., and Sadayappan, P. (2018b). Associative instruction reordering to alleviate register pressure. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 46:1–46:13, Piscataway, NJ, USA. IEEE Press.

[134] Rawat, P. S., Vaidya, M., Sukumaran-Rajam, A., Ravishankar, M., Grover, V., Rountev, A., Pouchet, L., and Sadayappan, P. (2018). Domain-specific optimization and generation of high-performance gpu code for stencil computations. *Proceedings of the IEEE*, 106(11):1902–1920.

[135] Rawat, P. S., Vaidya, M., Sukumaran-Rajam, A., Ravishankar, M., Grover, V., Rountev, A., Pouchet, L.-N., and Sadayappan, P. (2018). Domain-specific optimization and generation of high-performance gpu code for stencil computations. *Proceedings of the IEEE*, (99):1–19.

[136] Reference Guide, A. (2019). Graphics core next architecture, generation 3. http://developer.amd.com/wordpress/media/2013/12/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf. [Online; accessed 19-November-2019].

[137] Reinders, J. (2007). *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.".

[138] Rouhifar, M., Hosseinzadeh, M., Bahanfar, S., and Teshnehlab, M. (2011). Fast overflow detection in moduli set {2n–1, 2n, 2n+ 1}. *International Journal of computer science issues*, 8(3):407–414.

[139] Sakata, S. and Kurihara, M. (1997). A systolic array architecture for implementing a fast parallel decoding algorithm of one-point ag codes. In *Proceedings of IEEE International Symposium on Information Theory*, page 378. IEEE.

[140] Salaria, S., Drozd, A., Podobas, A., and Matsuoka, S. (2019). Learning neural representations for predicting gpu performance. In *International Conference on High Performance Computing*, pages 40–58. Springer.

[141] Schwiegelshohn, U. (1988). A shortperiodic two-dimensional systolic sorting algorithm. In *[1988] Proceedings. International Conference on Systolic Arrays*, pages 257–264. IEEE.

[142] Shimokawabe, T., Takaki, T., Endo, T., Yamanaka, A., Maruyama, N., Aoki, T., Nukada, A., and Matsuoka, S. (2011). Peta-scale phase-field simulation for dendritic solidification on the tsubame 2.0 supercomputer. In *High Performance Computing,*

Networking, Storage and Analysis (SC), 2011 International Conference for, pages 1–11. IEEE.

[143] Slomp, M. P. B. (2008). Real-time photographic local tone reproduction using summed-area tables.

[144] Sodani, A., Gramunt, R., Corbal, J., Kim, H.-S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., and Liu, Y.-C. (2016). Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2):34–46.

[145] Stone, J. E., Gohara, D., and Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66.

[146] Tang, W. T., Tan, W. J., Krishnamoorthy, R., Wong, Y. W., Kuo, S. H., Goh, R. S. M., Turner, S. J., and Wong, W. F. (2013). Optimizing and auto-tuning iterative stencil loops for gpus with the in-plane method. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 452–462.

[147] Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., and Leiserson, C. E. (2011). The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–128. ACM.

[148] Top500 (2019). TOP 10 Sites for June 2019. https://www.top500.org/lists/2019/06/. [Online; accessed 01-August-2019].

[149] Van Meel, J. A., Arnold, A., Frenkel, D., Portegies Zwart, S., and Belleman, R. G. (2008). Harvesting graphics power for md simulations. *Molecular Simulation*, 34(3):259–266.

[150] Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S., and LeCun, Y. (2014). Fast convolutional nets with fbfft: A gpu performance evaluation, 2014. *URL http://arxiv. org/abs/1412.7580*.

[151] Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., and Catthoor, F. (2013). Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23.

[152] Viola, P. and Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–I. IEEE.

[153] Waldrop, M. M. (2016). The chips are down for moore's law. *Nature News*, 530(7589):144.

[154] Wang, X., Han, T. X., and Yan, S. (2009). An hog-lbp human detector with partial occlusion handling. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 32–39. IEEE.

[155] Wei, X., Yu, C. H., Zhang, P., Chen, Y., Wang, Y., Hu, H., Liang, Y., and Cong, J. (2017). Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 29. ACM.

[156] Wikipedia (2019a). Arcade system board — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Arcade%20system%20board&oldid=904495412. [Online; accessed 23-July-2019].

[157] Wikipedia (2019b). Graphics processing unit — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Graphics%20processing%20unit&oldid=906001710. [Online; accessed 23-July-2019].

[158] Wikipedia (2019c). Prefix sum — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Prefix%20sum&oldid=908207191. [Online; accessed 01-August-2019].

[159] Wikipedia (2019d). Very Large Scale Integration — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Very%20Large%20Scale%20Integration&oldid=897049854. [Online; accessed 19-July-2019].

[160] Wikipedia (2019e). Video display controller — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Video%20display%20controller&oldid=902680687. [Online; accessed 23-July-2019].

[161] Wikipedia contributors (2019). Graphics core next — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Graphics_Core_Next&oldid=926002841. [Online; accessed 27-November-2019].

[162] Wilde, D. K. (1993). A library for doing polyhedral operations. Technical Report 785, IRISA.

[163] Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76.

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

[165] Woo, M., Neider, J., Davis, T., and Shreiner, D. (1999). *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc.

[166] Xi, M., Chen, L., Polajnar, D., and Tong, W. (2016). Local binary pattern network: a deep learning approach for face recognition. In *Image processing (ICIP), 2016 IEEE international conference on*, pages 3224–3228. IEEE.

[167] Xia, Y., Jiang, P., and Agrawal, G. (2019). Enabling prefix sum parallelism pattern for recurrences with principled function reconstruction. In *Proceedings of the 28th International Conference on Compiler Construction*, CC 2019, pages 17–28, New York, NY, USA. ACM.

[168] Yalamanchili, P., Arshad, U., Mohammed, Z., Garigipati, P., Entschev, P., Kloppenborg, B., Malcolm, J., and Melonakos, J. (2015). ArrayFire - A high performance software library for parallel computing with an easy-to-use API.

[169] Yeh, C.-S., Reed, I. S., and Truong, T.-K. (1984). Systolic multipliers for finite fields gf (2 m). *IEEE Transactions on Computers*, (4):357–360.

[170] Yuan, L., Zhang, Y., Guo, P., and Huang, S. (2017). Tessellating stencils. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 49:1–49:13.

[171] Zhang, G. and Zhao, Y. (2016). Modeling the Performance of 2.5D Blocking of 3D Stencil Code on GPUs. In *IEEE High Performance Extreme Computing Conference, HPEC*, pages 117–122. IEEE.

[172] Zhang, X., Tan, G., Xue, S., Li, J., Zhou, K., and Chen, M. (2017). Understanding the gpu microarchitecture to achieve bare-metal performance tuning. In *ACM SIGPLAN Notices*, volume 52, pages 31–43. ACM.

[173] Zhao, T., Williams, S., Hall, M., and Johansen, H. (2018). 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)*, pages 59–70. IEEE.

[174] Zohouri, H. R., Podobas, A., and Matsuoka, S. (2018). 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, FPGA 2018, Monterey, CA, USA, February 25-27, 2018*, pages 153–162.

[175] Zumbusch, G. (2012). Vectorized higher order finite difference kernels. In *International Workshop on Applied Parallel Computing*, pages 343–357. Springer.