A Versatile Software Systolic Execution Model for GPU Memory-Bound Kernels

Peng Chen
Tokyo Institute of Technology
AIST-Tokyo Tech Real World
Big-Data Computation Open
Innovation Laboratory, National
Institute of Advanced Industrial
Science and Technology
chen.p.aa@m.titech.ac.jp

Mohamed Wahib
AIST-Tokyo Tech Real World
Big-Data Computation Open
Innovation Laboratory, National
Institute of Advanced Industrial
Science and Technology
mohamed.attia@aist.go.jp

Shinichiro Takizawa
AIST-Tokyo Tech Real World
Big-Data Computation Open
Innovation Laboratory, National
Institute of Advanced Industrial
Science and Technology
shinichiro.takizawa@aist.go.jp

Ryousei Takano
National Institute of Advanced
Industrial Science and Technology
takano-ryousei@aist.go.jp

Satoshi Matsuoka
Tokyo Institute of Technology
RIKEN Center for Computational
Science, Hyogo, Japan
matsu@acm.org

ABSTRACT

This paper proposes a versatile high-performance execution model, inspired by systolic arrays, for memory-bound regular kernels running on CUDA-enabled GPUs. We formulate a systolic model that shifts partial sums by CUDA warp primitives for the computation. We also employ register files as a cache resource in order to operate the entire model efficiently. We demonstrate the effectiveness and versatility of the proposed model for a wide variety of stencil kernels that appear commonly in HPC, and also convolution kernels (increasingly important in deep learning workloads). Our algorithm outperforms the top reported state-of-the-art stencil implementations, including implementations with sophisticated temporal and spatial blocking techniques, on the two latest Nvidia architectures: Tesla V100 and P100. For 2D convolution of general filter sizes and shapes, our algorithm is on average 2.5× faster than Nvidia's NPP on V100 and P100 GPUs.

CCS CONCEPTS

• **Computer systems organization** → **Systolic arrays**; **Multi-core architectures**.

KEYWORDS

Systolic Array, GPU, CUDA, Convolution, Stencil

ACM Reference Format:
Peng Chen, Mohamed Wahib, Shinichiro Takizawa, Ryousei Takano, and Satoshi Matsuoka. 2019. A Versatile Software Systolic Execution Model for GPU Memory-Bound Kernels. In *SC '19, November 17–22, 2019, Denver, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

SC '19, November 17–22, 2019, Denver, USA

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

GPU accelerators have been increasingly adopted to meet the exponentially growing computational requirements in various fields, such as scientific simulations and machine learning. Those increasing computational requirements are pushing for the trend of building GPU-accelerated supercomputers made up of dense nodes that include several GPUs (e.g. ORNL Summit has six GPUs/node, and LLNL's Sierra and TokyoTech's Tsubame 3.0 both have four GPUs/node). Hence it is imperative that codes running on those systems be high in performance and scale vertically (i.e. on a single node), as well as horizontally (i.e. on the entire system). In this paper, we focus on the single node performance of one of 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 [31, 32, 36, 41, 52, 58]). 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: register files.

This paper proposes a versatile systolic execution model for improving the performance of memory-bound kernels with regular

access patterns. A wide class of kernels and applications can benefit from this versatile model: convolution, stencils, scan/reduction operators [35], Summed Area Tables [7], ...etc. The systolic model is based on the transfer and accumulation of partial results in thread-private registers. Additionally, we employ the register files as a cache to avoid using the scratchpad altogether.

To accumulate and transfer partial sums using thread-private registers, in a SIMT fashion, different groups of threads (known as warps in CUDA) operate over different input points, with some data redundancy that we introduce to account for the halo layers. Different threads in a warp compute the partial sums, before moving the partial sums to the downstream neighbor thread to be accumulated. To transfer the partial sums, we rely on the warp shuffle primitives that provide low-latency register exchange within a warp [43].

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*.

A systolic array model [33] is typically a well-structured two-dimensional mesh of Processing Elements (PEs) and provides extremely high TOPS (TeraOps/Second) and high TOPS/Watt [17]. Recently, systolic arrays have been successfully used to speed up deep learning workloads, e.g. Google Tensor Processing Unit (TPU) [17] and Nvidia Tensor Cores. Inspired by the mechanism of hard-wired systolic arrays, we propose *a versatile execution model* to improve the performance of regular memory-bound kernels by moving the memory access bottleneck from the scratchpad to registers. Our model can be viewed as Software Systolic Array Execution Model (SSAM).

Kernels that can be mapped to SSAM should have a regular memory access pattern. However, there are several challenges that should be addressed in order 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. The contributions in this paper are as follows:

- A formulation, design, and implementation of a model (called SSAM), inspired by systolic arrays, for efficiently computing memory-bound kernels with regular access on GPUs [2].
- 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 Tesla P100/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 over 2.5× faster than Nvidia's NPP library, and up to 1.5× faster than the ArrayFire library.

The rest of this paper is organized as follows. Section 2 discusses the background, i.e., CUDA, convolution, and stencils. In Section 3 we propose the formulation of SSAM. In Section 4 we present the implementation of various algorithms in SSAM. Section 5 describes our performance model. In Section 6 we report the evaluated performance of convolution and stencils on the latest Nvidia GPUs. Section 7 discusses the GPU architectures from the perspective of operation dependencies in SSAM. In Section 8, we review the related work. Finally, Section 9 concludes.

2 BACKGROUND

We briefly introduce the CUDA's concepts and programming model (more details about CUDA can be found in [44]): (i) **CUDA Memory Hierarchy**. GPUs support different memory types: global, local, texture, constant, shared and register files [44]. Global memory is the largest off-chip memory with the highest R/W (Read/Write) latency. Shared memory is a fast on-chip scratchpad memory limited in scope to CUDA thread blocks. (ii) **Register Cache Vs. Shared Memory Cache.** Register cache is an approach in which a single warp builds a virtual cache layer on top of register files for low-latency data R/W [4]. As Table 1 shows, on the latest GPUs, the register memory per SM is 256KB (65536*4B) and is more than 2.7× larger than shared memory. (iii) **Intra-Warp Communication By Shuffle.** Shuffle is an intra-warp communication mechanism for a CUDA-enabled GPU. It allows the exchange of data between threads directly within a single warp without using shared memory or global memory. (iv) **Register Spilling.** Registers are a limited resource in GPUs. If the required number of registers per thread is too high, the compiler may spill to local memory [37]. However, there are methods to reduce the register pressure as in [48, 51].

2.1 Convolution

Mathematically, convolution combines two linear functions to form a third one in order to measure the correlation of overlap between functions. 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 (or image), while w is a filter of size (M, N) applied to the matrix, where M=b-a+1, N=d-c+1. Hereafter, we assume that the size of the matrix is (W, H) where W is the width of the matrix and H is its height.

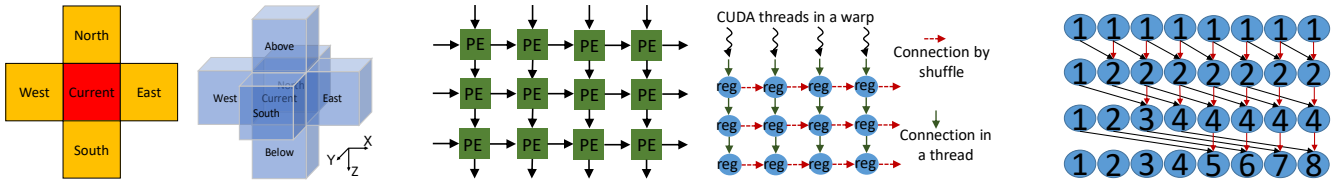
Note that throughout the paper, we use W, H, M, N as defined in this section.

2.2 Stencils

Iterative stencil computations are fundamental for scientific applications in many domains [19, 38]. Figure 1a shows a typical

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

[2] The source code and experiments for SSAM publicaly avilable at https://github.com/pengdada/ssam-sc19.



(a) 2D stencil. (b) 3D stencil. (c) Hardware Systolic Array. (d) Software Systolic Array. (e) KS-scan for 8 elements [20].

Figure 1: (a) 2D 5-point stencil example. (b) 3D 7-point stencil example. (c) Hardware 2D systolic array structure. PE is a processing element. (d) SSAM on CUDA (2D problem illustration: "reg" is a register). In the vertical direction, registers are in the same thread. In the horizontal direction, registers are exchanged by the shuffle instruction. (e) Eight elements KS-scan [20]. Arrows represent the dependency of our model in Equation 2.

Table 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

first-order 2D diffusion 5-point stencil (also known as 2D Jacobi stencil). Mathematically, the 2D diffusion stencil is defined as

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

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.

Regarding 3D stencils, taking the 7-point diffusion stencil as an example (Figure 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.

3 SSAM: SOFTWARE SYSTOLIC ARRAY MODEL

We propose an execution model, named Software Systolic Array Execution Model (SSAM), that enhances the performance of regular memory-bound kernels on GPUs. Our work is motivated by the recent revival of systolic arrays and similar architectures such as Tensor cores of Nvidia GPUs, Intel TBB [50] data flow graphs, and wavefront accelerators (mainly FPGAs [18] and ASICs [6]). Tensor cores are specially optimized for matrix multiplication, however, SSAM is a general model that is applicable to a wide range of memory-bound algorithms.

3.1 Systolic Arrays

A systolic array [25] is a structured computing model composed of many interconnected, yet independent, Processing Elements (PEs) as shown in Figure 1c. 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

$$s \leftarrow ctrl(r \otimes x) \oplus s \quad (1)$$

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.

3.2 Hardware Systolic Array

In the past decades, systolic arrays have been well researched. The authors in [8, 23, 29, 46, 57] have proposed well-structured systolic arrays architectures. Ideally, a systolic array is qualified for a specific computing purpose: e.g., matrix multiplication [22], convolution [24], DCT [27] ...etc. A hardware systolic array is a computing pipeline network with physical interconnects between PEs. As Figure 1c shows, all of the PEs are connected with their neighbors. The PEs simultaneously compute, store, and transfer their partial results to downstream neighbors.

3.3 SSAM: Software Systolic Array Model

SSAM is built as an execution model to express a variety of algorithms, with regular access pattern, in a systolic array fashion. In other words, it simulates the mechanism of hardware systolic arrays in CUDA architecture. Our target in this paper is to improve the performance of the memory-bound kernels with regular memory access pattern. 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:

(i) Efficient in-register computation via register cache;
(ii) Fast intra-warp communication via shuffle instructions;
(iii) 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 1d shows, each register performs the same function as a PE in Figure 1c. 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.

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 8). 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.

3.4 Expressing Algorithms in SSAM

In order to express algorithms in SSAM, we build an algorithmic formulation that extends the prior work in the literature [39]. From the perspective of a CUDA warp, an algorithm can be formulated as a four-tuple

$$\mathcal{J} = (O, D, X, Y) \tag{2}$$

where

- O : computing operations of $\mathcal{J}$
- D : dependencies of $\mathcal{J}$
- X : input variables of $\mathcal{J}$
- Y : output variables of $\mathcal{J}$

In the model $\mathcal{J}$, the register cache is used for storing X and Y (more details about register cache will be discussed in Section 4.2). In contrast to hardware systolic arrays, SSAM can perform computing operations O like Equation 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 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 7.2 : using the polyhedral model [12] to represent D.

3.5 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 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 1) where r∈F, $\otimes$ and $\oplus$ are multiplication and addition operations, respectively. The dependency graph D can be represented as in Figure 2c, where the $ctrl() \equiv 1$ is fixed.

3.6 Motivating Example 2: Scan Operator

Scan operator is the sums of prefixes (running total) of an input sequence [13, 20]. 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 1), $r \equiv 1$, $\otimes$ is a multiplication and $\oplus$ is an addition operation. Since we use the Kogge-Stone Scan algorithm [20], the dependency graph D is similar to Figure 1e, namely the arrows in Figure 1e can be expressed as *ctrl()* in Equation 1.

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, more details can be found in our another work [7].

4 IMPLEMENTATION OF ALGORITHMS 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.

4.1 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 $\mathcal{J}$ (Equation 2), the input X and output Y are addressed in Section 4.2. The computing operation O is discussed in Section 4.3, and the dependency graph D is discussed in Section 4.4. The detailed 2D convolution implemented by SSAM model is shown in Listing 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).
(iii) As shown in Figure 2a, according to the sliding window[3] 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).

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

Listing 1: SSAM-based 2D Convolution: CUDA kernel. M, N, W, H are defined in Section 2.1, P, B and C are illustrated in Section 4.2.

```
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[DATA_IDX] = src[SRC_IDX];
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[DST_IDX] = data[DATA_IDX];
32 }
```

(v) We move the sliding window step by step for a total of P times (line 17) in Listing. 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.

4.2 Register Cache & Coalesced Memory Access

In SSAM, and in 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 1. The operation is repeated to cache multiple lines of data from global memory into register files, line by line. As illustrated in Figure 2a, each thread in a single warp caches C elements as

$$C = N + P - 1 \tag{3}$$

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 2a, 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 2b. Next, we shift the partial inner product to the neighboring threads as shown in Figure 2c. It is worth emphasizing that such a sliding window provides a simple yet effective method to tackle the second challenge in Section 1.

4.3 Computing Partial Sums

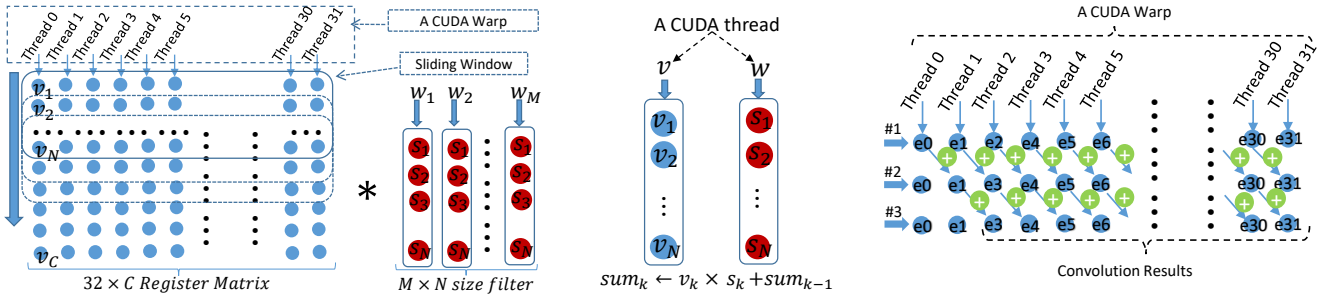
In this section, we describe the computation of partial sums. Computing the partial sum for 2D convolution by Equation 1 is similar to the 1-D convolution example in Section 3.5. More specifically as seen in Figure 2b, all threads simultaneously, in a systolic fashion, compute the partial sum (sum_k) 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 2b. 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 [44]). For the M×N filter, the inner products are computed M times (as shown in Figure 2a) to compute a single output element of convolution (Listing 1 line 20~27).

4.4 Transferring & Accumulating Partial Sums

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 2a 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 2c, all of the registers are shifted only once at each step (Listing 1 line 22). Next, the shifted partial results are added to the accumulated results by each thread (Listing 1 line 25). This process is repeated *M-1* times (Listing 1 line 20~27). Finally a row of convolution results could be attained from a group of threads, namely whose *laneIds* [44] ranges from *M-1* to *WarpSize-1*. By moving the sliding window once, each warp of threads computes *WarpSize-M+1* convolution results.

4.5 Overlapped Blocking Scheme

The overlapped blocking scheme is widely adopted to improve the concurrency in time-tiled stencils computations [21, 62]. We use this scheme to eliminate warp-divergence, which can have negative performance effects. As Figure 3 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



(a) Register Cache. (b) Computing Partial Sums. (c) Transfer Partial Sums.

Figure 2: (a) Computing 2D convolution by a single warp to 32×C register-cached words and M×N filter matrix. (b) An example of computing partial sums in parallel. (c) Shifting partial results: 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. ⊕ is an addition operator.

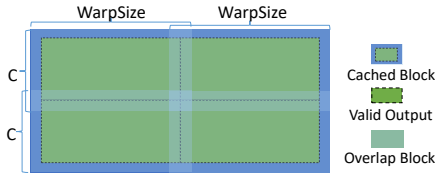


Figure 3: Overlapped Blocking: the cached block size is WarpSize×C, the valid output size is (WarpSize-M+1)×P. Overlapped block size is a function of the specified filter size.

and most complex part of computing the 2D convolution (Listing 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 (the last challenge in Section 1). We provide an elaborate analysis of the halo layers impact on performance (performance model in Section 5.3).

4.6 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 [44].

4.7 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

Listing 2: 2D 5-point stencil CUDA kernel. Variable definitions are the same as Listing 1. *West, North, Current, South* and *East* are stencil coefficients.

```
1  #pragma unroll
2  for (int i=0; i<P; i++) {
3     T sum = 0;
4      sum = MAD(data[i + 1], West, sum);  //1th column
5      sum = __shfl_up_sync(0xffffffff, sum, 1);
6      sum = MAD(data[i + 0], North, sum); //2th column
7      sum = MAD(data[i + 1], Current, sum);
8      sum = MAD(data[i + 2], South, sum);
9      sum = __shfl_up_sync(0xffffffff, sum, 1);
10     sum = MAD(data[i + 1], East, sum);  //3th column
11     data[i] = sum;
12 }
```

$WarpCount = B/WarpSize$. The required CUDA grid dimensions are expressed as $GridDim.x = \lceil \frac{W}{WarpCount \cdot (WarpSize-M+1)} \rceil$ and $GridDim.y = \lceil H/P \rceil = \lceil H/(C-N+1) \rceil$, respectively.

4.8 Mapping 2D Stencil to SSAM

In this section, we describe mapping 2D stencils to SSAM. In Listing 2, the SSAM-based 5-point stencil kernel differs slightly from the convolution example in Listing 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. 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.

4.9 Mapping 3D Stencil to SSAM

This section discusses using SSAM for a 3D stencil. We divide the 3D grid into many 3D sub-grids with overlapping blocks (to account for the halo layers as in Figure 3). Each sub-grid is processed

Table 2: The latency of different operations. 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.

GPU	Operation	Latency	GPU	Operation	Latency
P100	shfl_up_sync	33	V100	shfl_up_sync	22
	add, sub, mad	6		add, sub, mad	4
	T_{smem_read}	33		T_{smem_read}	27

by a CUDA block, and each warp in a CUDA block processes a 2D slice in the X-Y plane as in Figure 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 (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.

5 PERFORMANCE MODEL

In this section, we introduce a performance model to analyze the effectiveness of SSAM over the conventional scratchpad memory implementations. In addition, we analyze the overhead of processing the halo area required in the overlapped blocking scheme (in Section 4.5). 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.

5.1 Micro-benchmarking

We use micro-benchmarking to better understand some performance characteristics of GPUs. Several papers have made contributions in demystifying parts of the details about CUDA-enabled GPUs, such as the memory hierarchy latency and throughput [34]. We adopt the micro-benchmarks proposed by cudabmk [55], and improve its functions for our purpose. Some of the relevant measured results are listed in Table 2.

5.2 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 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

$$L_{reg} = M{\cdot}N{\cdot}(T_{mad} + T_{smem\_read} + 2{\cdot}T_{reg}) + (M-1){\cdot}T_{shfl} \quad (4)$$

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

$$\begin{aligned} Dif_{smem\_reg} &= L_{smem} - L_{reg} \\ &= M{\cdot}N{\cdot}T_{smem\_read} - (M-1){\cdot}T_{shfl} \end{aligned} \quad (5)$$

Based on the metrics in Table 2, 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.

5.3 Halo Layer(s) Overhead

In this section, we analyze the benefits and overhead of using overlapped blocking (as shown in Section 4.5) 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 3 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

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

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

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

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∼400 cycles/warp for coalesced access [42]. We can conclude that $AvgDif \gg 0$, where M≥2 and N≥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.

5.4 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 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 1d 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.

6 EVALUATION

This section is dedicated to reporting the achieved performance of our execution model using the latest Nvidia GPUs. The performance of 2D convolution and 2D/3D stencil computation is analyzed in detail. It is noteworthy to mention that we prioritized the implementation and reporting of 3D stencil over 3D convolution in the paper, in part due to 3D stencils being more exhaustively studied in literature and targeted with complex data reuse schemes (shown later). We are however currently targeting 3D convolutions after performing an exhaustive study of the diversity of 3D convolutions sizes used in deep learning.

6.1 Software & Hardware Setup

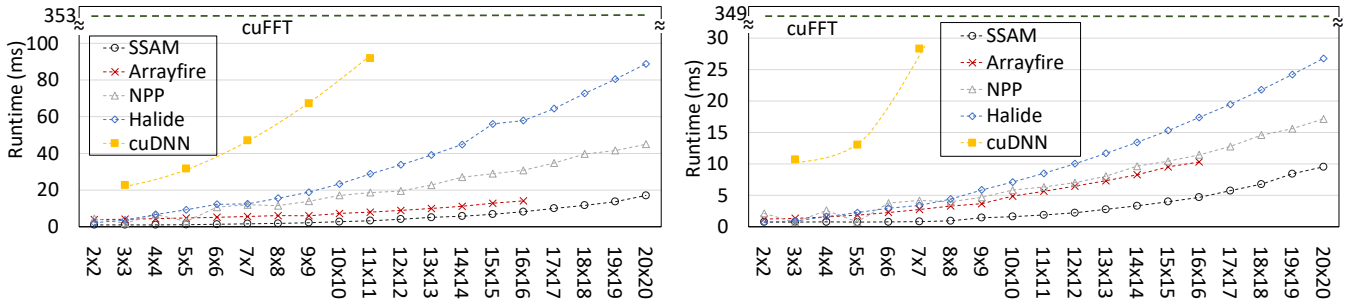
The experimental results presented here are evaluated by two Tesla P100 and V100 GPUs. The CUDA driver is 410.48, we use CUDA 10.0 (including NPP/cuFFT and cuDNN v7.4), GPU memory is 16GB, and the OS is CentOS 7.6. ECC option for GPUs is on. Nvidia nvcc and gcc-5.4 are used to compile the CUDA kernel and host codes, respectively.

6.2 2D Convolution Results

In Figure 4, we evaluate 2D convolutions for various filter sizes, which ranges from 2×2 to 20×20. 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** [56]. 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** [44]. 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** [44]. The Nvidia CUDA Fast Fourier Transform (cuFFT) library provides a constant, yet relatively high, run time regardless of the filter size.
(iv) **Halide** [40]. The Halide is a domain-specific language (DSL) designed to generates pipelines for image processing [40]. 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** [9]. 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 gives indications of the accuracy of the findings of the performance model (Section 5). Equation 5 indicates that given a fixed output size of 2D convolution, while increasing size of filters, the performance difference between SSAM and other kinds of implementations (not only shared memory-based algorithm)



(a) Single precision on P100. cuFFT method is constant as 353 ms.

(b) Single precision on V100. cuFFT method is constant as 349 ms.

Figure 4: 2D Convolution performance and scalability. The image size is 8192×8192, the x-axis is filter size, the y-axis is execution time.

Table 3: Stencil benchmark. The k is stencil order, 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 [47, 48].

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

becomes increasingly larger. Such a conclusion can be verified by observing the difference in the performance of the graphs in Figure 4.

6.3 Stencil Results

To better understand the performance of SSAM-based stencil computations, we evaluate a diverse collection of 2D/3D stencil benchmark (listed in Table 3) 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 order stencils since high order stencils are typically bound by the registers. For instance, temporal blocking is effective for low order stencils [62], while register re-ordering schemes are effective for high order stencils [47, 48].

Figure 5 shows three implementations: "*original*", "*reordered*", and "*unrolled*", with state-of-the-art performance results by Rawat et al. [47, 48]. 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 [53] and Halide [40].

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 3). Note that the execution time of all stencil kernels is measured by Nvidia *nvprof* profiler.

In Figure 5, 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) and is further explained by the performance model.

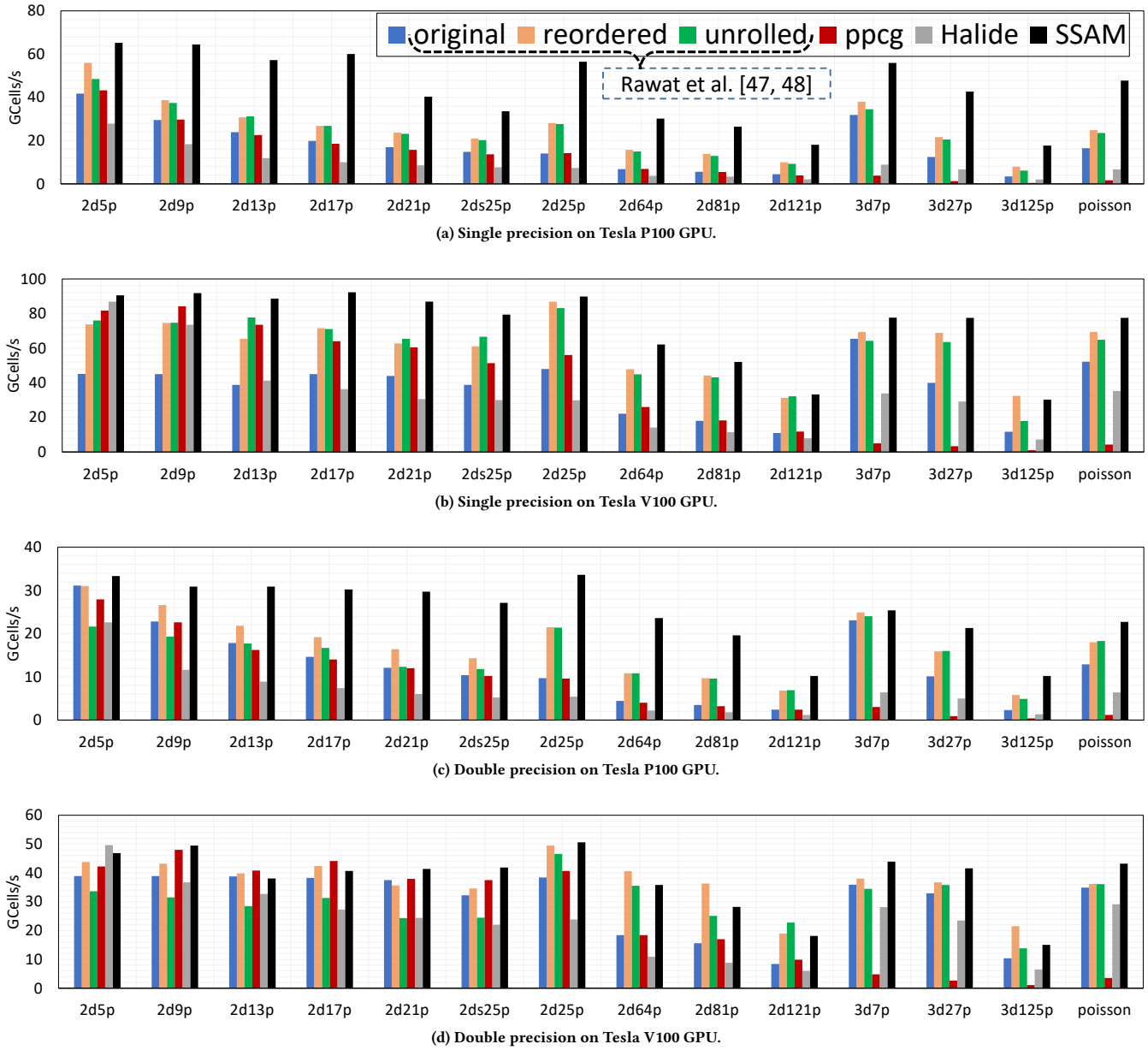
There are two important observations in Figure 5. One is that in comparison to the P100 GPU, the performance variance on V100 become smaller, and the other is that when using double precision on the V100 GPU, few of the SSAM-based stencils do not achieve the highest performance. The reason behind that is that Volta's architecture is different in many aspects, which will be discussed in Sec 7.1.

6.4 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** [49]. To the author's knowledge, StencilGen reports the highest performance among temporal blocking libraries. As Figure 6 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. In addition, 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 [48, 49], can further be adopted by SSAM to improve the performance for the specific cases of deep temporal blocking.

Diffusion. The authors in [62] report a competitive performance of the 3d7pt stencil (highly optimized by shared memory as proposed in [32], called **Diffusion** in Figure 6) as 92.7 GCells/s



(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.

Figure 5: 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 3. 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 [47, 48], the "ppcg" is auto-generated from the C code using the ppcg compiler [53].

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 [61]. 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 6a and Figure 6b, SSAM outperforms Bricks on P100. Since Bricks is not publicly available, we can not compare our result for the V100 GPU.

7 DISCUSSION

This section discusses the performance considerations given to Pascal and Volta architectures, and elaborates on how to extract dependency for SSAM.

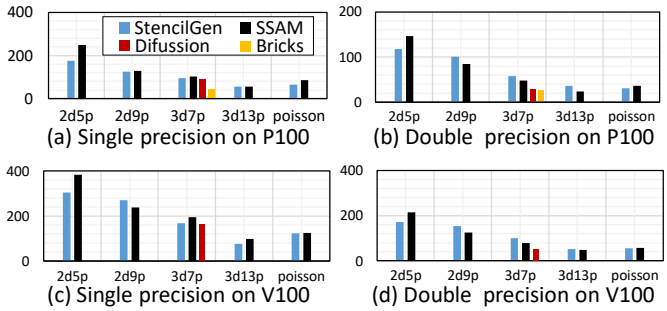


Figure 6: Performance evaluation for the 2D and 3D stencil computation using temporal blocking. the x-axis is stencil benchmark, the y-axis is the performance in the unit of GCells/s.

7.1 Considerations for Pascal & Volta Architectures

In this section, we outline the reasons why SSAM performs differently on P100 and V100 GPUs.

(i) The L1 cache in Volta is significantly enhanced. (1) According to Nvidia user guide [45], 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 [15]: 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 of 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 [15], 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 ≤234 cycles vs. V100 is ≤193 cycles).

(iii) As Jia et al. [16] 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, 60]. Finally, SSAM performs as well as Pascal in Maxwell and Kepler architectures. Due to the space limitation, we do not show the result.

7.2 Automation and Extracting Dependency (D) for SSAM

This paper proposes a method for executing kernels with regular data-access behavior in a systolic fashion. The paper 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 Section 6 are strictly implemented as Listing 1 and Listing 2 without extra manual optimization. Taking Figure 4 for instance, Listing 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 code bases 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 paper due to the space limit, 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 2) using the polyhedral model [12, 30, 54]. The polyhedral model is a well-researched mathematical framework for performance optimization [5], which often involves nested loops and large numbers of operations, i.e. convolution, matrix multiplication. According to our formulation in Equation 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 RELATED WORK

Register cache methods are widely used on GPUs to boost individual application's performance [4, 11, 14, 26]. Most notably, a technology called register packing demonstrated how to avoid shared memory communication for improving cyclic reduction performance [10]. Lai and Seznec [26] suggest a method named register blocking to explore the potential peak performance of SGEMM on Fermi and Kepler GPUs. Enfedaque et al. proposed a discrete wavelet transform (DWT) implementation for fast image processing based on register cache [11]. The binary finite field multiplication algorithm was implemented by Eli Ben-Sasson et al. yielded up to 138× speedup than the popular Number Theory Library [4]. Hou et al. [14] implemented a register-based sort method shows great improvements over scratchpad memory methods on NVIDIA K80-Kepler and TitanX-Pascal GPUs. A 1-D stencil method is introduced as an example to illustrate how register cache and shuffle instruction works [4].

Register optimization is an important approach to improve the performance of stencil codes. The authors in [59] implemented a register-only method to improve the 3D 7-point stencil without using thread communication. Zumbusch et al. implemented vectorized kernels for high order finite stencils on multi-platforms, i.e. AMD/Intel CPUs, Nvidia GPUs [63]. Rawat et al. proposed a reorder framework to optimize register allocation for both CPUs and GPUs [47, 48].

Partial sums method is used by Basu et al. to generate SIMD code for CPUs within CHiLL compiler [3]. On the other hand, SSAM is introduced at the application level to perform the in-register computation and not just partial sums for GPUs. Krishnamoorthy

et al. used *overlapped tiling* optimization for the stencil to preserve concurrency in time-tiled computations [21].

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. 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 [28].

StencilGen [49], a state-of-the-art DSL implementing advanced temporal blocking for stencils. Halide [40] is a DSL and ppcg [53] 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 earlier sections.

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 register 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.

9 CONCLUSION AND FUTURE WORK

This paper proposes a novel and versatile high-performance execution model (namely SSAM) for improving the performance of structured grid memory-bound kernels on CUDA-enabled GPUs. Using the SSAM execution model, we improve the performance of challenging problems, such as convolution and stencil. Our model performs as a software systolic array by computing and shifting partial sums. The evaluation shows that SSAM outperforms top libraries for 2D convolution and 2D/3D stencil, in both Pascal and Volta GPU architectures.

Nvidia Volta V100 GPU includes hardware systolic arraya (namely Tensor cores). Tensor cores are efficient in computation yet they are limited in precision since they designed for Deep Learning workloads. Our software systolic array can be used for a varity of applications at single and double precision. For future work, we plan to apply our model to 3D/4D convolution workload for accelerating deep learning training. We also intend to generate SSAM codes automatically using polyhedral analysis and DSL tools, e.g. ppcg, Halide.

ACKNOWLEDGMENT

This work was partially supported by JST-CREST under Grant Number JPMJCR1687. Computational resource of AI Bridging Cloud Infrastructure (ABCI) provided by National Institute of Advanced Industrial Science and Technology (AIST) was used. A part of the numerical calculations were carried out on the TSUBAME3.0 supercomputer at Tokyo Institute of Technology. We would like to thank Endo Lab at Tokyo Institute of Technology for providing a Tesla V100 GPU Server.

REFERENCES

[1] M. J. Anderson, D. Sheffield, and K. Keutzer. 2012. A Predictive Model for Solving Small Linear Algebra Problems in GPU Registers. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 2–13. https://doi.org/10.1109/IPDPS.2012.11

[2] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. 2009. A View of the Parallel Computing Landscape. *Commun. ACM* 52, 10 (Oct. 2009), 56–67.

[3] P. Basu, M. Hall, S. Williams, B. V. Straalen, L. Oliker, and P. Colella. 2015. Compiler-Directed Transformation for Higher-Order Stencils. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 313–323. https://doi.org/10.1109/IPDPS.2015.103

[4] Eli Ben-Sasson, Matan Hamilis, Mark Silberstein, and Eran Tromer. 2016. Fast Multiplication in Binary Fields on GPUs via Register Cache. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 35, 12 pages. https://doi.org/10.1145/2925426.2926259

[5] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*. Springer, 283–303.

[6] Himanshu Bhatnagar. 2002. *Advanced ASIC chip synthesis*. Springer.

[7] 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)*. IEEE, 482–493.

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

[9] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[10] Andrew Davidson and John D. Owens. 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)*. ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/1964179.1964185

[11] Pablo Enfedaque, Francesc Auli-Llinas, and Juan C Moure. 2015. Implementation of the DWT in a GPU through a register-based strategy. *IEEE Transactions on Parallel and Distributed Systems* 26, 12 (2015), 3394–3406.

[12] Martin Griebl, Christian Lengauer, and Sabine Wetzel. 1998. Code Generation in the Polytope Model. In *In IEEE PACT*. IEEE Computer Society Press, 106–111.

[13] Mark Harris, Shubhabrata Sengupta, and John D Owens. 2007. Parallel prefix sum (scan) with CUDA. *GPU gems* 3, 39 (2007), 851–876.

[14] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. 2017. Fast segmented sort on GPUs. In *Proceedings of the International Conference on Supercomputing*. ACM, 12.

[15] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. *arXiv preprint arXiv:1903.07486* (2019).

[16] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).

[17] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 1–12.

[18] Steve Kilts. 2007. *Advanced FPGA design: architecture, implementation, and optimization*. John Wiley & Sons.

[19] Peter E Kloeden and RA Pearson. 1977. The numerical solution of stochastic differential equations. *The ANZIAM Journal* 20, 1 (1977), 8–12.

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

[21] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. 2007. Effective Automatic Parallelization of Stencil Computations. *SIGPLAN Not.* 42, 6 (June 2007), 235–244. https://doi.org/10.1145/1273442.1250761

[22] VK Prasanna Kumar and Y-C Tsai. 1991. On synthesizing optimal family of linear systolic arrays for matrix multiplication. *IEEE Trans. Comput.* 40, 6 (1991), 770–774.

[23] HT Kung. 1979. Let's design algorithms for VLSI systems. (1979).

[24] HT Kung and RL Picard. 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*. 273–278.

[25] Hsiang-Tsung Kung. 1982. Why systolic architectures? *IEEE computer* 15, 1 (1982), 37–46.

[26] Junjie Lai and André Seznec. 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)*. IEEE Computer Society, 1–10.

[27] Moon Ho Lee, Jong Oh Park, and Yasuhiko Yasuda. 1990. Simple systolic arrays for discrete cosine transform. *Multidimensional Systems and Signal Processing* 1, 4 (1990), 389–398.

[28] Ang Li, Weifeng Liu, Linnan Wang, Kevin Barker, and Shuaiwen Leon Song. 2018. Warp-Consolidation: A Novel Execution Model for GPUs. In *Proceedings of the 2018 International Conference on Supercomputing (ICS '18)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/3205289.3205294

[29] Guo-Jie Li et al. 1985. The design of optimal systolic arrays. *IEEE Trans. Comput.* 100, 1 (1985), 66–77.

[30] Vincent Loechner. 1999. PolyLib: A library for manipulating parameterized polyhedra. https://repo.or.cz/polylib.git/blob_plain/HEAD:/doc/parampoly-doc.ps.gz

[31] Tareq M. Malas, Georg Hager, Hatem Ltaief, Holger Stengel, Gerhard Wellein, and David E. Keyes. 2015. Multicore-Optimized Wavefront Diamond Blocking for Optimizing Stencil Updates. *SIAM J. Scientific Computing* 37, 4 (2015).

[32] Naoya Maruyama and Takayuki Aoki. 2014. Optimizing Stencil Computations for NVIDIA Kepler GPUs. In *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*, Armin Größlinger and Harald Köstler (Eds.). Vienna, Austria, 89–95.

[33] JV McCanny, JG McWhirter, and K Wood. 1984. Optimised bit level systolic array for convolution. In *IEE Proceedings F-Communications, Radar and Signal Processing*, Vol. 131. IET, 632–637.

[34] Xinxin Mei, Kaiyong Zhao, Chengjian Liu, and Xiaowen Chu. 2014. Benchmarking the memory hierarchy of modern GPUs. In *IFIP International Conference on Network and Parallel Computing*. Springer, 144–156.

[35] Duane Merrill. 2015. Cub: Cuda unbound. *URL: http://nvlabs. github. io/cub* (2015).

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

[37] Paulius Micikevicius. 2011. Local memory and register spilling. *NVIDIA Corporation* (2011).

[38] William Edmund Milne and WE Milne. 1953. *Numerical solution of differential equations*. Vol. 19. Wiley New York.

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

[40] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 83.

[41] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 2010. 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*. 1–13.

[42] Nvidia. 2017. CUDA C PROGRAMMING GUIDE. (2017). https://docs.nvidia.com/cuda/archive/9.1/pdf/CUDA_C_Programming_Guide.pdf

[43] Nvidia. 2017. NVIDIA TESLA V100 GPU ARCHITECTURE. *Technical whitepaper. https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf* (2017).

[44] Nvidia. 2019. CUDA Toolkit Documentation. *NVIDIA Developer Zone. https://docs.nvidia.com/cuda/index.html* (2019).

[45] Nvidia. 2019. Tuning CUDA Applications for Volta. *https://docs.nvidia.com/cuda/volta-tuning-guide/index.html* (2019).

[46] Patrice Quinton. 1983. *The systematic design of systolic arrays*. Ph.D. Dissertation. INRIA.

[47] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2018. Register Optimizations for Stencils on GPUs. *SIGPLAN Not.* 53, 1 (Feb. 2018), 168–182. https://doi.org/10.1145/3200691.3178500

[48] Prashant Singh Rawat, Aravind Sukumaran-Rajam, Atanas Rountev, Fabrice Rastello, Louis-Noël Pouchet, and P. Sadayappan. 2018. Associative Instruction Reordering to Alleviate Register Pressure. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 46, 13 pages. http://dl.acm.org/citation.cfm?id=3291656.3291718

[49] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L. Pouchet, and P. Sadayappan. 2018. Domain-Specific Optimization and Generation of High-Performance GPU Code for Stencil Computations. *Proc. IEEE* 106, 11 (Nov 2018), 1902–1920. https://doi.org/10.1109/JPROC.2018.2862896

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

[51] Putt Sakdhnagool, Amit Sabne, and Rudolf Eigenmann. 2019. RegDem: Increasing GPU Performance via Shared Memory Register Spilling. arXiv:arXiv:1907.02894

[52] W. T. Tang, W. J. Tan, R. Krishnamoorthy, Y. W. Wong, S. H. Kuo, R. S. M. Goh, S. J. Turner, and W. F. Wong. 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*. 452–462.

[53] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[54] Doran K. Wilde. 1993. *A Library for Doing Polyhedral Operations*. Technical Report 785. IRISA.

[55] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 235–246.

[56] Pavan Yalamanchili, Umar Arshad, Zakiuddin Mohammed, Pradeep Garigipati, Peter Entschev, Brian Kloppenborg, James Malcolm, and John Melonakos. 2015. ArrayFire - A high performance software library for parallel computing with an easy-to-use API. https://github.com/arrayfire/arrayfire

[57] C-S Yeh, Irving S. Reed, and Trieu-Kien Truong. 1984. Systolic multipliers for finite fields GF (2 m). *IEEE Trans. Comput.* 4 (1984), 357–360.

[58] Liang Yuan, Yunquan Zhang, Peng Guo, and Shan Huang. 2017. Tessellating Stencils. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. Article 49, 13 pages.

[59] Guangwei Zhang and Yinliang Zhao. 2016. Modeling the Performance of 2.5D Blocking of 3D Stencil Code on GPUs. In *IEEE High Performance Extreme Computing Conference, HPEC*. IEEE, 117–122.

[60] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU microarchitecture to achieve bare-metal performance tuning. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 31–43.

[61] Tuowen Zhao, Samuel Williams, Mary Hall, and Hans Johansen. 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)*. IEEE, 59–70.

[62] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 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*. 153–162.

[63] Gerhard Zumbusch. 2012. Vectorized higher order finite difference kernels. In *International Workshop on Applied Parallel Computing*. Springer, 343–357.