Implementing Strassen's Algorithm with CUTLASS on NVIDIA Volta GPUs

FLAME Working Note #88

Jianyu Huang[*†], Chenhan D. Yu[*†], Robert A. van de Geijn[*†]
[*]Department of Computer Science
[†]Institute for Computational Engineering and Sciences
The University of Texas at Austin, Austin, TX 78712
{jianyu.huang@, chenhan@, rvdg@cs.}utexas.edu

August 23, 2018

Abstract

Conventional GPU implementations of Strassen's algorithm (STRASSEN) typically rely on the existing high-performance matrix multiplication (GEMM), trading space for time. As a result, such approaches can only achieve practical speedup for relatively large, "squarish" matrices due to the extra memory overhead, and their usages are limited due to the considerable workspace. We present novel STRASSEN primitives for GPUs that can be composed to generate a family of STRASSEN algorithms. Our algorithms utilize both the memory and thread hierarchies on GPUs, reusing shared memory and register files inherited from GEMM, fusing additional operations, and avoiding extra workspace. We further exploit intra- and inter-kernel parallelism by batching, streaming, and employing atomic operations. We also develop a performance model for NVIDIA Volta GPUs to select the appropriate blocking parameters and predict the performance for GEMM and STRASSEN. Overall, our 1-level STRASSEN can achieve up to 1.11× speedup with a crossover point as small as 1,536 compared to `cublasSgemm` on a NVIDIA Tesla V100 GPU. With additional workspace, our 2-level STRASSEN can achieve 1.19× speedup with a crossover point at 7,680.

1 Introduction

Given matrices $A \in \mathcal{R}^{m\times k}$, $B \in \mathcal{R}^{k\times n}$, and $C \in \mathcal{R}^{m\times n}$, Strassen's algorithm (STRASSEN) [31] computes matrix multiplication (GEMM defined in `BLAS` [8] and `cuBLAS` [27])

$$C = \alpha A \times B + \beta C \tag{1}$$

with less than $\mathcal{O}(n^3)$ work. The algorithm partitions the matrices into 2×2 submatrices such that

$$\begin{bmatrix} C_0 & C_1 \\ C_2 & C_3 \end{bmatrix} = \alpha \begin{bmatrix} A_0 & A_1 \\ A_2 & A_3 \end{bmatrix} \begin{bmatrix} B_0 & B_1 \\ B_2 & B_3 \end{bmatrix} + \beta \begin{bmatrix} C_0 & C_1 \\ C_2 & C_3 \end{bmatrix}, \tag{2}$$

and rearrange the arithmetic operations to reduce the number of submatrix multiplications from 8 to 7 (see Section 3 for details). By recursively applying this scheme, it can be shown [31] that (1) only requires $\mathcal{O}(n^{2.81})$ work.

Although it is easy to observe the saving from the complexity analysis, the achievable practical speedup is typically disappointing due to the extra memory overhead and space requirement [4, 6, 14] (see Figure 1). A recent paper [14] addresses these issues and provides a good review on the related work on modern CPU architectures. We extend the idea in [14] and present a new STRASSEN algorithm on GPUs.

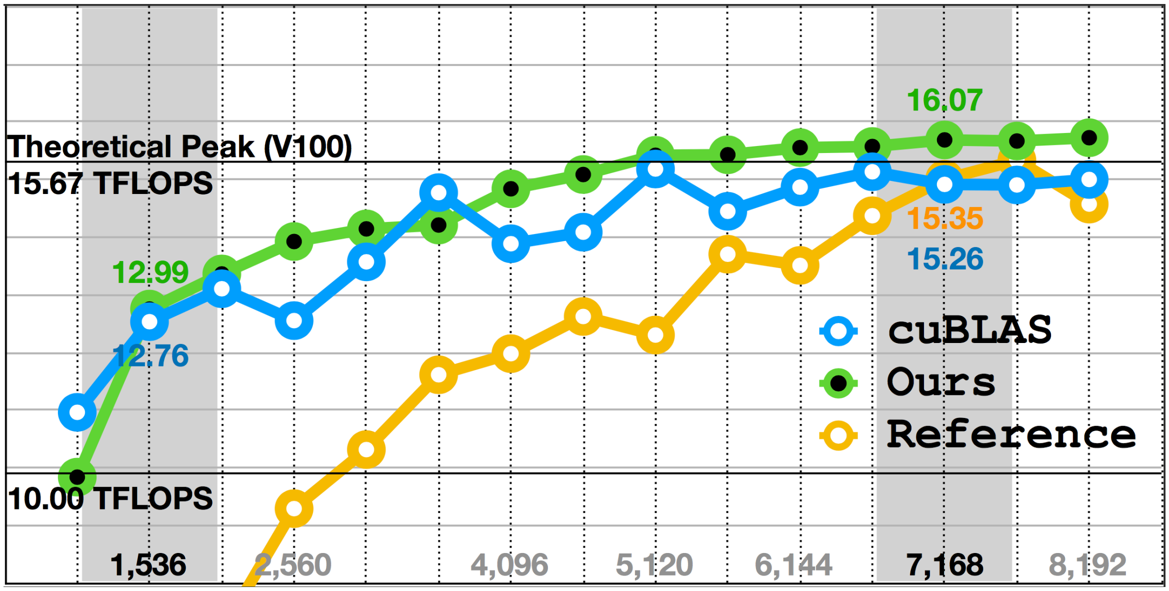

Figure 1: Break-even point of our Strassen implementation and the state-of-the-art [20]: the **x-axis** denotes the problem size ($m = n = k$), and the **y-axis** denotes the floating point operation efficiency in `TFLOPS`. For a square matrix-multiplication, this work can achieve speedup over `cublasSgemm` for problem size as small as 1,536 while the state-of-the-art requires at least 7,168 to break even (10k is required to obtain a stable speedup).

Challenges: A practical Strassen implementation on GPUs must overcome several challenges. First, the GPU architecture and programming model are different from their counterparts for a CPU. In order to achieve high performance, a practical implementation of Strassen needs to leverage the memory and thread hierarchies on GPUs. Second, a GPU has a limited physical memory capacity. The conventional Strassen implementations require some extra temporary memory for storing intermediate submatrices, which limit the maximum problem size that can be computed compared to GEMM because of the GPU memory capacity. Third, a GPU is a highly parallel, multi-thread, many-core processor. Strassen needs to be parallelized at multiple granularities to fully utilize the computational horsepower of GPUs. There is thus a tension between reducing the memory and exploiting more parallelism with the conventional implementation of Strassen. Finally, the ratio between the theoretical peak performance and memory bandwidth of a GPU is even higher (less favorable) than that of a CPU. Strassen has a lower ratio of arithmetic operations to memory operations compared to GEMM, which means Strassen only becomes advantageous when the problem sizes are sufficiently large. As a result, the practical implementation of Strassen needs to reduce the extra data movement to save the bandwidth and outperform GEMM for small or moderate problem sizes.
Contributions: Inspired by [14] and the recent development of `CUTLASS` [17] (reviewed in Section 2.3), we introduce new algorithms for the practical implementation of Strassen on GPUs. To be specific,

- We develop new GPU Strassen kernels (Section 3.1), which fuse additional memory and arithmetic operations with the GEMM pipeline. As a result, no additional workspace (GPU global memory and shared memory) is required.

- We present and discuss different optimization schemes and generate different kernels that effectively reduce the number of required registers (Section 3.2).

- Our algorithms exploit both intra- and inter-kernel task-based parallelism. This allows us to maintain the parallelism without increasing the kernel launching and context switching overhead (Section 3.3).

- We derive an accurate performance model on NVIDIA Volta GPUs, which can help us to choose the right blocking parameters and predict the performance for GEMM and Strassen (Section 5).

- We conduct experiments on different matrix shapes (Section 4). For square cases, our 1-level STRASSEN has a break-even point (faster than `cublasSgemm`) as small as 1,536, while the state-of-the-art requires at least 7,168. Our hybrid 2-level STRASSEN has a break-even point as small as 7,680, while the state-of-the-art requires at least 13,312. Our implementations are also more efficient for non-square cases.

Limitation: While the proposed approach does not require extra workspace (in the global memory), it still trades memory operations (`mops`) for floating point operations (`flops`). As a result, it may not be the optimal algorithm, and extra space is preferred to offload the increasing register requirement and the global memory latency while applying STRASSEN algorithms to multiple levels. This trade-off is discussed in Section 5. Furthermore, STRASSEN is known to experience degradation in numerical stability, especially when more than two levels of recursions are incorporated [11, 7, 2]. For this reason, only a few levels of recursions are leveraged.

Related work: The literature on the theory and practice of STRASSEN is vast. For a review, see [14]. To our knowledge, there are no practical STRASSEN implementations on GPUs [21, 20] that can be free from extra workspace and have a break-even point as small as 1,536. The only algorithm and software that comes close is [20], which still requires additional $\mathcal{O}(mk/4 + kn/4 + mn/4)$ space. In Section 4, we also provide empirical results with this algorithm as a reference. The idea of operation-fusing has also been generalized to other domains to effectively reduce slow memory operations (improve temporal locality of the cache hierarchy) and extra space requirement in tensor contraction and other N-body operations. For a review, see [23, 30, 12, 35]. High-performance GEMM implementations on GPUs can be found in [24, 34, 19, 10, 32, 36].

2 Background

In this section, we first briefly review the GPU programming model (CUDA) and the GPU architecture (Volta) we use in this paper. We then review the state-of-the-art algorithm (the `CUTLASS` framework) of high-performance GEMM on GPUs.

2.1 GPU Programming Model

The CUDA programming model [25] assumes that the CUDA program (*kernel*) is executed on physically independent *devices* (GPUs) as coprocessors to the *host* (CPU). Figure 2 shows the memory and thread hierarchies on the GPU device.

Memory hierarchy: The memory hierarchy on the GPU device includes three levels: global memory, shared memory (co-located with L1 and texture caches [28]), and register files. The latency decreases while the bandwidth increases through the memory hierarchy from global memory to registers.

Thread hierarchy: A *thread* is the smallest execution unit in a CUDA program. A *thread block* is a group of threads that run on the same core and shares a partition of resources such as shared memory. Thread blocks communicate through barrier synchronization. Multiple blocks are combined to form a *grid*, which corresponds to an active CUDA kernel on the device. At runtime, a thread block is divided into a number of *warps* for execution on the cores. A warp is a set of 32 threads to execute the same instructions while operating on different data in lockstep.

2.2 NVIDIA Volta GPUs

We review the hardware specification of the NVIDIA Tesla V100 [9], which features a GV100 (Volta) microarchitecture. Tesla V100 is comprised of 80 streaming multiprocessors (SMs). Each SM is partitioned into 4 processing blocks. Each processing block consists of 2 Tensor Cores, 8 `FP64` (double precision) cores, 16 `FP32` (single precision) cores, and 16 `INT32` cores. The tested Tesla V100 SXM2 GPU accelerator has the base clock frequency 1.3 GHz and boost clock frequency 1.53 GHz. As a result, the theoretical peak performance can reach 15.67 `TFLOPS`[1] with single precision and 7.83 `TFLOPS`[2] with double precision, while

[1] 1 `FMA`/cycle × 2 `flop/FMA` × 1.53G (boost clock frequency) × 16 (# `FP32` core) × 4 (# processing block/SM) × 80 (# SM).

[2] 1 `FMA`/cycle × 2 `flop/FMA` × 1.53G (boost clock frequency) × 8 (# `FP64` core) × 4 (# processing block/SM) × 80 (# SM).

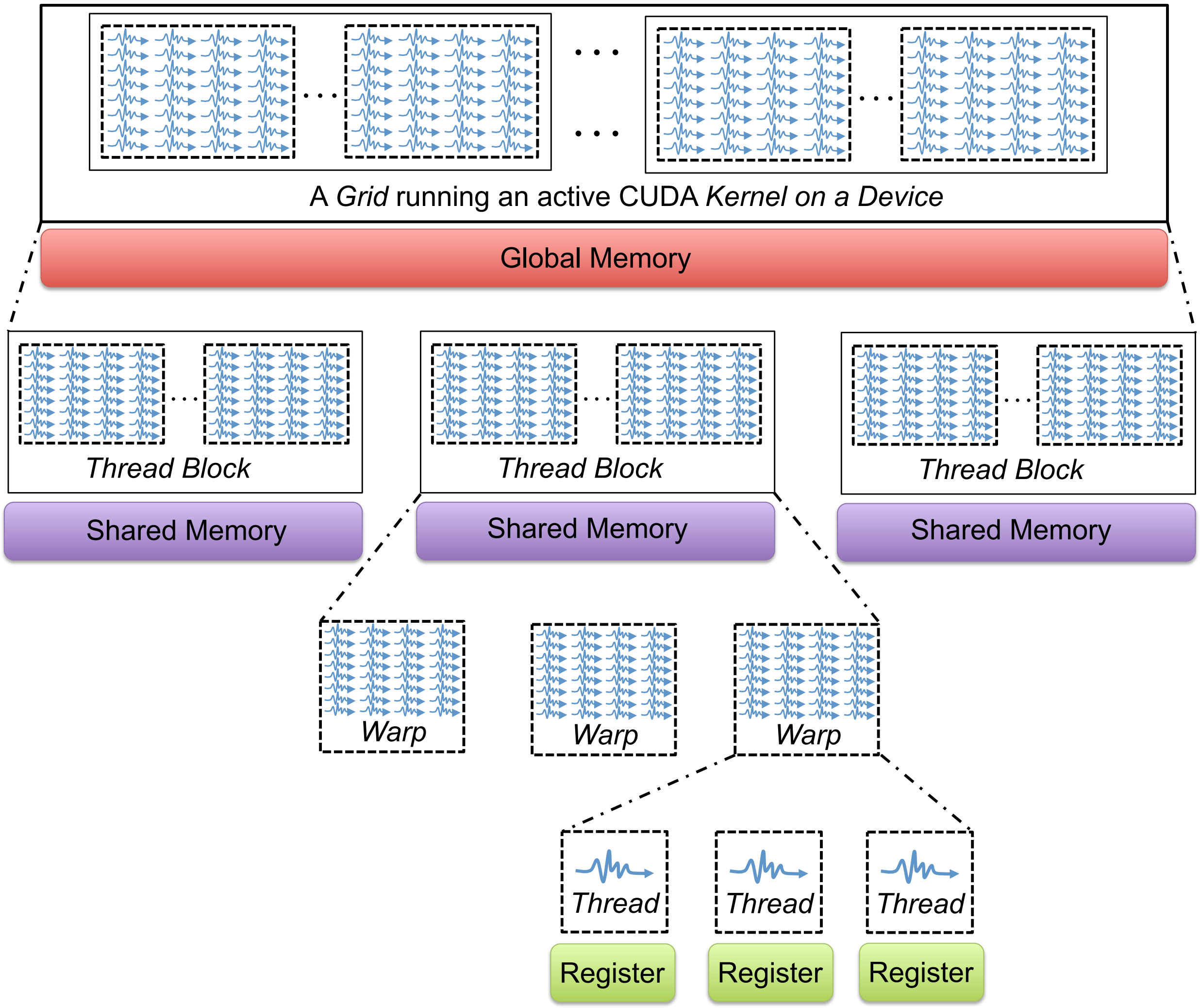

Figure 2: The memory and thread hierarchies in the CUDA programming model.

Tensor Cores can deliver 125 TFLOPS[3] for FP16/FP32 mixed precision. The tested Tesla V100 GPU is built using 16 GB HBM2 memory with 900 GB/s of bandwidth.

2.3 Matrix Multiplication on GPUs

We review the high-performance implementation of GEMM on NVIDIA GPUs, based on NVIDIA's CUDA Templates for Linear Algebra Subroutines (CUTLASS) [17, 5], a collection of CUDA C++ templates and abstractions to instantiate high-performance GEMM operations. CUTLASS incorporates strategies for hierarchical partition and data movement similar to cuBLAS [27], the state-of-the-art implementation of the BLAS implementation on NVIDIA GPU, and can reach more than 90% of cuBLAS performance on V100. Without loss of generality, we will focus on single precision arithmetic and $\alpha = \beta = 1$ in (1) henceforth.

2.3.1 Blocking Strategies

Figure 3 illustrates the GEMM implementation in CUTLASS. It organizes the computation by partitioning the operands into blocks in the different levels of the device, thread block, warp, and thread.

Device Level: blocking for the thread blocks and shared memory. The three operand matrices, A, B, and C, are partitioned into $m_S \times k_S$, $k_S \times n_S$, and $m_S \times n_S$ blocks. Each thread block computes an $m_S \times n_S$ block of C by accumulating the results of matrix products of an $m_S \times k_S$ block of A and a $k_S \times n_S$ block of B. Therefore, the $m_S \times n_S$ block of C (the output of the thread block) is referred as the *C Accumulator*. Since it is updated many times, it needs to be lifted into the fastest memory in the SM: the register files. The global memory corresponding to the *C Accumulator* only needs to be updated once after the *C Accumulator* has accumulated the results of all matrix products along with the k dimension. Furthermore, to improve data locality, blocks of A and B are "*packed*" (copied) from global memory into shared memory (the *A Tile* and *B Tile*) for data reuse, accessible by all threads in the same thread block.

Thread Block Level: blocking for the warps. After the *A Tile* and *B Tile* are stored in shared memory, each individual warp computes a sequence of accumulated outer products by iteratively loading an *A Fragment* (a subcolumn of the *A Tile* with height m_W) and a *B Fragment* (a subrow of the *B Tile* with width n_W) from the corresponding shared memory into register files along the k dimension and performing a rank-1 update. The *C Accumulator* is spatially partitioned across all the warps within the same thread block, with each warp storing a non-overlapping 2-D block in the register files.

Warp Level: blocking for the threads. Each thread in a warp computes an $m_R \times n_R$ outer product with subvectors of the *A Fragment* and subvectors of the *B Fragment* in a "strip-mining" (cyclic) pattern. Each piece has a size of 4, because the largest granularity of vector load is 128 bits (4 single precision floating point numbers), and this helps to maximize the effective bandwidth. The total length of all pieces for an individual thread in m dimension is m_R, while the total length in n dimension is n_R. Since each warp has 32 threads, CUTLASS organizes the threads within the same warp in a 4 × 8 or 8 × 4 fashion such that $m_W/m_R = 4$, $n_W/n_R = 8$, or $m_W/m_R = 8$, $n_W/n_R = 4$.

Thread Level: executing on the CUDA cores. Each thread issues a sequence of independent FMA instructions to the CUDA cores and accumulates an $m_R \times n_R$ outer product.

2.3.2 Choices of Block Sizes

CUTLASS customizes six different strategies of block sizes at each level $\{m_S, n_S, k_S, m_R, n_R, m_W, n_W\}$ in Figure 3 for different matrix shapes and sizes, as shown in Figure 4. Details about how to choose these blocking parameters for large problem sizes are given in Section 5.2. Note that each thread block has $m_S/m_R \times n_S/n_R$ threads.

2.3.3 Software Prefetching

As shown in the left of Algorithm 1, to keep the SM busy, CUTLASS uses global and local software prefetching to hide the data movement latency. The computations on the CUDA cores are overlapped with the data

[3] 64 FMA/cycle × 2 flop/FMA × 1.53G (boost clock frequency) × 2 (# Tensor Core) × 4 (# processing block/SM) × 80 (# SM).

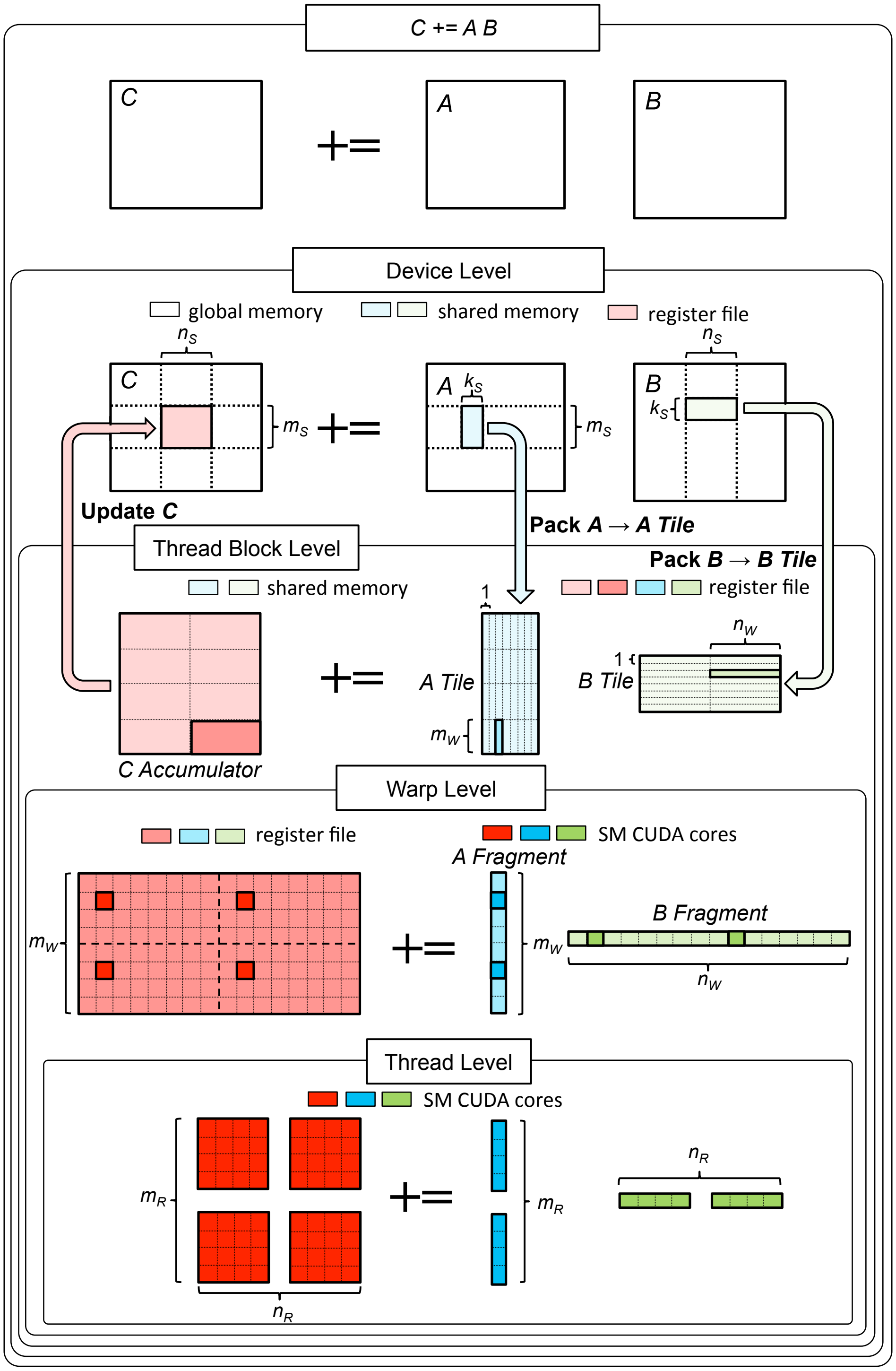

Figure 3: Illustration of the GEMM implementation in CUTLASS [17]. CUTLASS partitions the operand matrices into blocks in the different levels of the device, thread block, warp, and thread. Here we show block sizes typical for the large SGEMM: $m_S = 128$, $n_S = 128$, $k_S = 8$; $m_W = 4 \times m_R = 32$, $n_W = 8 \times n_R = 64$; $n_R = 8$, $n_R = 8$.

Strategy	m_S	n_S	k_S	m_R	n_R	m_W/m_R	n_W/n_R
Small	16	16	16	2	2	4	8
Medium	32	32	8	4	4	4	8
Large	64	64	8	8	8	4	8
Tall	128	32	8	8	4	8	4
Wide	32	128	8	4	8	4	8
Huge	128	128	8	8	8	4	8

Figure 4: `CUTLASS` specifies six strategies of block sizes at each level in Figure 3 for different matrix shapes and sizes.

preloading from the global memory (line 12 and 14 in Algorithm 1) and from the shared memory (line 17 and 18). A synchronization (line 22) is required to ensure that all shared memory writes to $tile_A$ and $tile_B$ between line 20 and 21 have completed before reading their values between line 12 and 14 in the next iteration[4].

3 Method

Following [14], if the three operands A, B, and C in (1) are partitioned into quadrants as in (2), then

$$\begin{array}{lll}
\textcircled{0} & M_0 = (A_0 + A_3)(B_0 + B_3); & C_0 {+}{=} M_0; C_3 {+}{=} M_0; \\
\textcircled{1} & M_1 = (A_2 + A_3)B_0; & C_2 {+}{=} M_1; C_3 {-}{=} M_1; \\
\textcircled{2} & M_2 = A_0(B_1 - B_3); & C_1 {+}{=} M_2; C_3 {+}{=} M_2; \\
\textcircled{3} & M_3 = A_3(B_2 - B_0); & C_0 {+}{=} M_3; C_2 {+}{=} M_3; \\
\textcircled{4} & M_4 = (A_0 + A_1)B_3; & C_1 {+}{=} M_4; C_0 {-}{=} M_4; \\
\textcircled{5} & M_5 = (A_2 - A_0)(B_0 + B_1); & C_3 {+}{=} M_5; \\
\textcircled{6} & M_6 = (A_1 - A_3)(B_2 + B_3); & C_0 {+}{=} M_6;
\end{array} \tag{3}$$

compute $C \mathrel{+}= AB$, with seven instead of eight (sub)matrix multiplications, decreasing the total number of arithmetic operations by a factor of 7/8 (ignoring total number of extra additions, a lower order term). If all matrices are square and of size $N \times N$, theoretically this single step of Strassen [31] can be applied recursively, resulting in the classical Strassen with a cost of $\mathcal{O}(N^{2.81})$.

The operations above in (3) are all instances of the following general Strassen primitive

$$M = (X + \delta Y)(V + \epsilon W); \quad D {+}{=} \gamma_0 M; \quad E {+}{=} \gamma_1 M; \tag{4}$$

with $\gamma_0, \gamma_1, \delta, \epsilon \in \{-1, 0, 1\}$. Here, X and Y are submatrices of A, V and W are submatrices of B, and D and E are submatrices of C. This scheme can be extended to multiple levels of Strassen [14].

We present a new GPU kernel that computes (4) in Section 3.1. We discuss how to effectively reduce the register requirement and generate different kernel variants in Section 3.2. Task parallelism is discussed in Section 3.3. Two-level Strassen algorithms and fringe case handling are discussed in Sections 3.4 and 3.5.

3.1 Strassen's Algorithm on NVIDIA GPUs

We extend the gemm implementation for GPUs illustrated in Figure 3 to accommodate the Strassen primitive

$$M = PQ = (X + Y)(V + W); D {+}{=} M; E {+}{=} M. \tag{5}$$

[4] `CUTLASS` also provides the option of double buffering on the thread block level to enable concurrent reading for the current iteration and writing for the next iteration. It eliminates the synchronization but also doubles the cost of the shared memory and the number of registers to hold the global memory fetches. On the Tesla V100 GPUs, the option of double buffering on the thread block level is disabled.

Algorithm 1 Comparisons between $C{+}{=}\ AB$ and $M = (X+Y)(V+W); D{+}{=}\ M; E{+}{=}\ M$ on GPUs with software prefetching

```
01:Register:  $frag_A[2][m_R], frag_B[2][n_R]$
02:Register:  $next0_A[m_R], next0_B[n_R]$
03:NOP
04:Register:  $accum_C[m_R \times n_R]$
05:Shared memory:  $tile_A[k_S \times m_S], tile_B[k_S \times n_S]$
06:load one $m_S \times k_S$ of $A$ into $tile_A[k_S][m_S]$
07:load one $k_S \times n_S$ of $B$ into $tile_B[k_S][n_S]$
08:__syncthreads()
09:load subvectors of first column in $tile_A$ into $frag_A[0][m_R]$
10:load subvectors of first row in $tile_B$ into $frag_B[0][n_R]$
11:for $block\_k = 0 : k_S : k$ then
12:    prefetch one subcolumn of next $m_S \times k_S$ block of $A$ into $next0_A[m_R]$
13:    NOP
14:    prefetch one subrow of next $k_S \times n_S$ block of $B$ into $next0_B[n_R]$
15:    NOP
16:    for $warp\_k = 0 : 1 : k_S$ then
17:        prefetch next subcolumns in $tile_A$ into $frag_A[(warp\_k+1)\%2][m_R]$
18:        prefetch next subrows in $tile_B$ into $frag_B[(warp\_k+1)\%2][n_R]$
19:        $accum_C[m_R][n_R]$ += $frag_A[warp\_k\%2][m_R] frag_B[warp\_k\%2][n_R]$
20:    store $next_A[m_R]$ into $tile_A[k_S][m_S]$
21:    store $next_B[n_R]$ into $tile_B[k_S][n_S]$
22:    __syncthreads()
23:write back $accum_C[m_R][n_R]$ to $m_S \times n_S$ block of $C$
24:NOP
```

∞

```
01:Register:  $frag_A[2][m_R], frag_B[2][n_R]$
02:Register:  $next0_A[m_R], next0_B[n_R]$
03:Register:  $next1_A[m_R], next1_B[n_R]$
04:Register:  $accum_C[m_R \times n_R]$
05:Shared memory:  $tile_A[k_S \times m_S], tile_B[k_S \times n_S]$
06:load the sum of one $m_S \times k_S$ of $X$ and corresponding $m_S \times k_S$ of $Y$ into $tile_A[k_S][m_S]$
07:load the sum of one $k_S \times n_S$ of $V$ and corresponding $k_S \times n_S$ of $W$ into $tile_B[k_S][n_S]$
08:__syncthreads()
09:load subvectors of first column in $tile_A$ into $frag_A[0][m_R]$
10:load subvectors of first row in $tile_B$ into $frag_B[0][n_R]$
11:for $block\_k = 0 : k_S : k$ then
12:    prefetch one subcolumn of next $m_S \times k_S$ block of $X$ into $next0_A[m_R]$
13:    ($\delta$) prefetch one subcolumn of next $m_S \times k_S$ block of $Y$ into $next1_A[m_R]$
14:    prefetch one subrow of next $k_S \times n_S$ block of $V$ into $next0_B[n_R]$
15:    ($\epsilon$) prefetch one subrow of next $k_S \times n_S$ block of $W$ into $next1_B[n_R]$
16:    for $warp\_k = 0 : 1 : k_S$ then
17:        prefetch next subcolumns in $tile_A$ into $frag_A[(warp\_k+1)\%2][m_R]$
18:        prefetch next subrows in $tile_B$ into $frag_B[(warp\_k+1)\%2][n_R]$
19:        $accum_C[m_R][n_R]$ += $frag_A[warp\_k\%2][m_R] frag_B[warp\_k\%2][n_R]$
20:    ($\delta$) store $next0_A[m_R] + next1_A[m_R]$ into $tile_A[k_S][m_S]$
21:    ($\epsilon$) store $next0_B[n_R] + next1_B[n_R]$ into $tile_B[k_S][n_S]$
22:    __syncthreads()
23:write back $accum_C[m_R][n_R]$ to $m_S \times n_S$ block of $D$
24:($\gamma_1$) write back $accum_C[m_R][n_R]$ to $m_S \times n_S$ block of $E$
```

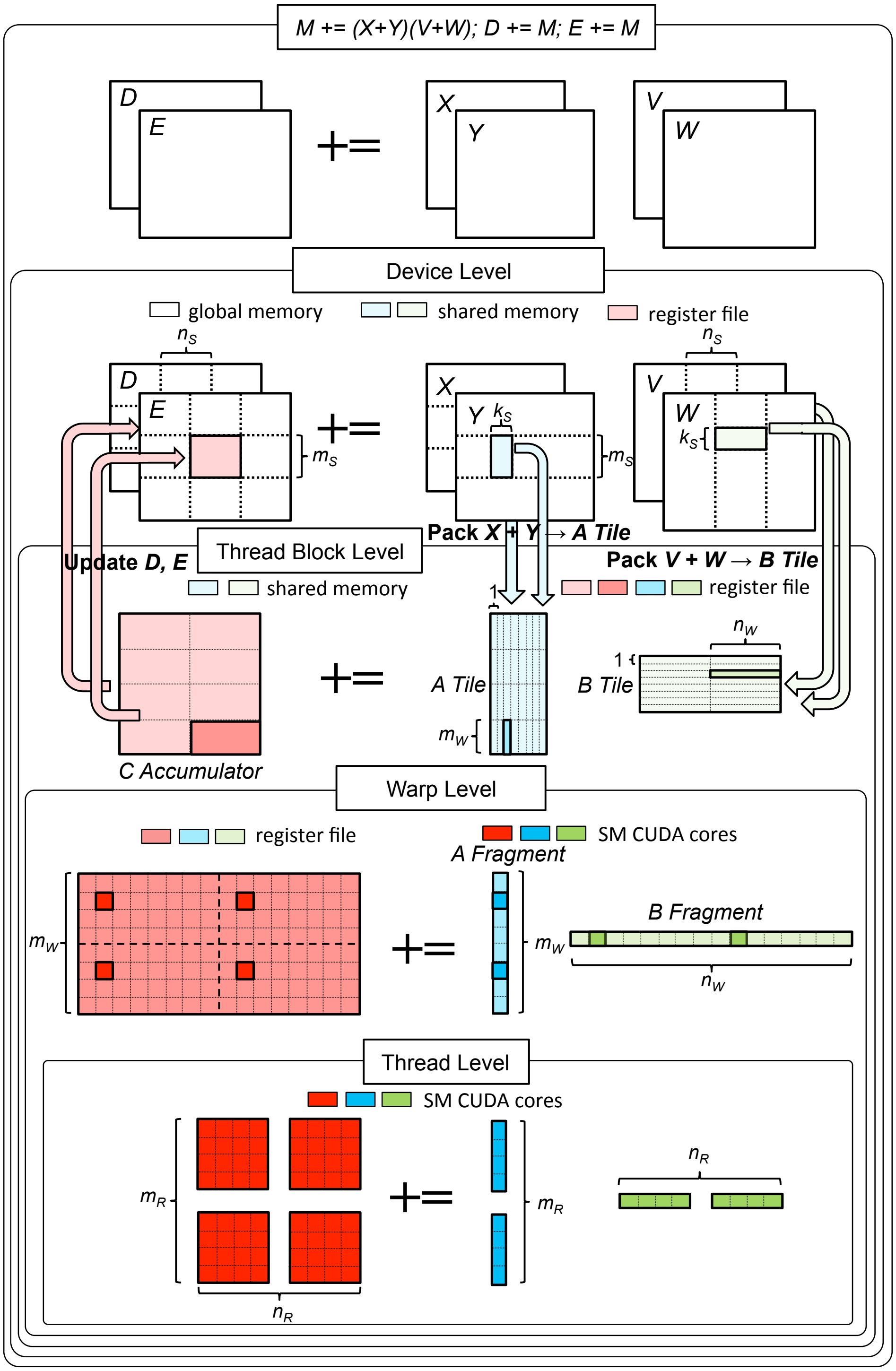

Figure 5: Specialized kernel that implements the representative computation $M = (X + Y)(V + W); D += M; E += M$ of each row of computations in (3) based on Figure 3. X, Y are submatrices of A; V, W are submatrices of B; D, E are submatrices of C; M is the intermediate matrix product.

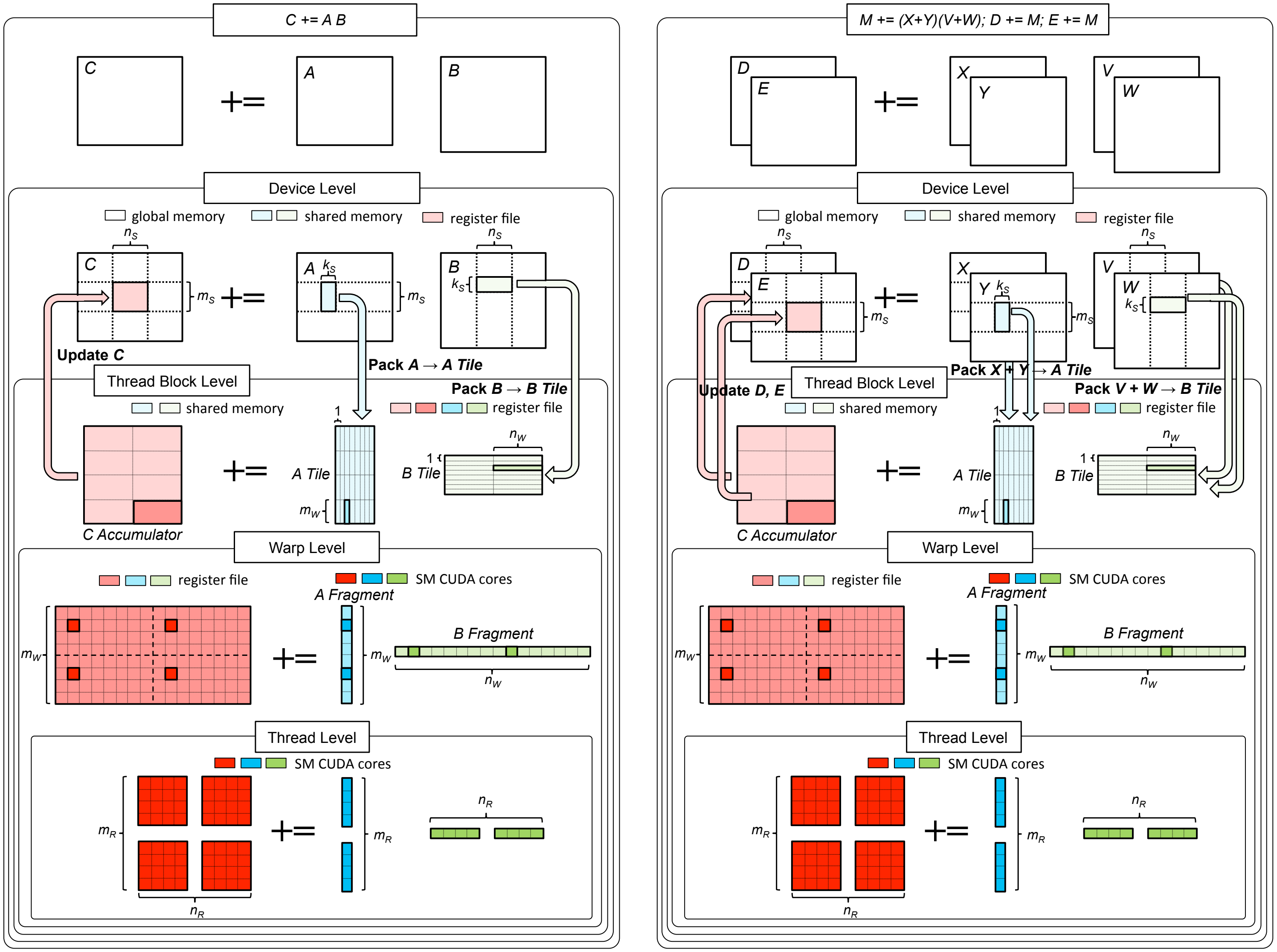

Figure 6: A side-by-side comparison of the GEMM implementation in CUTLASS and our modifications for implementing the representive computation $M = (X+Y)(V+W); D += M; E += M$. Left: Figure 3; Right: Figure 5.

		1-level				2-level									
Var#	*	0	1	2	3	0	1	2	3	4	5	6	7	8	9
W_A	1	2	2	1	2	4	1	2	4	4	4	4	2	2	4
W_B	1	2	2	2	1	4	4	4	1	2	4	4	2	4	2
W_C	1	2	1	2	2	4	4	4	4	4	1	2	4	2	2
Cnt#	1	1	2	2	2	1	4	4	4	4	4	4	8	8	8

Figure 7: Operand and instance counts of GEMM, 1-level, and 2-level STRASSEN primitives. The stared (*) column denotes the base case GEMM, which only has one operand per matrix. 1-level STRASSEN primitives have at most two operands per matrix, and overall there are 7 instances with 4 different variants. 2-level STRASSEN primitives have at most four operands per matrix, and overall there are 49 instances with 10 different variants.

The conventional approach performs pre-processing on the inputs $P = (X + Y)$, $Q = (V + W)$, and post-processing on its outputs $D{+}{=}\ M$ and $E{+}{=}\ M$. In other words, *the conventional approach must introduce extra workspace (in the global memory) and memory operations for intermediate matrices P, Q, and M to cast (5) in terms of calls to* GEMM.

Instead of casting the primitive in terms of GEMM, we develop a specialized kernel utilizing the memory and thread hierarchies on GPUs and show how these pre-processing and post-processing phases can be efficiently incorporated without introducing extra workspace. We illustrate how these extra memory operations (and a few floating point operations) are fused in Figure 5 and Figure 6 (right) without affecting the implementations in the warp and thread level:

Packing the A and B *Tiles*: The summation of matrices $X + Y$ can be incorporated into the packed A *Tile* during the packing process (from the **Device Level** to the **Thread Block Level** in Figure 6), avoiding the extra workspace requirement and reducing the additional memory movement since the A *Tile* is reused for the temporary matrix sum, which is held in the shared memory. Similarly, the summation of matrices $V + W$ can be also incorporated into the packed B *Tile* during the packing process.

Writing back the C *Accumulator*: After the C *Accumulator* has accumulated its result of $(X+Y)(V+W)$ along the k dimension, it can update the appropriate parts of D and E in the global memory once (from the **Thread Block Level** to the **Device Level**). This optimization avoids the required workspace for intermediate matrices M_i and reduces the additional memory movement since the C *Accumulator* is kept in the register files: it is fetched from the global memory into the register once in the beginning, and it is written to D and E only after its computation completes.

3.2 Register Optimization

We give the implementation of the STRASSEN primitive (on the right) side-by-side with `CUTLASS`'s GEMM algorithm (on the left) in Algorithm 1. Recall that the primitive incorporates pre-processing and post-processing steps to create a new kernel that avoids additional workspace. As a result, we must (for the current NVIDIA GPU architecture) introduce extra registers at line 3, extra `mops` at line 13, line 15, line 24, and extra `flops` at line 20 and line 21.

Notice that the algorithm presented in Algorithm 1 is the general form of the seven instances in (3). Depending on the value of scalars δ, ϵ and γ_1 (represented as statement predicates in Algorithm 1), we can generate specialized kernels at compile time (using `C++` non-type template parameters) that may optimize out these extra registers, `mops` and `flops`. Overall, there are four different variants (**Var#0**–**Var#3**) for the one-level STRASSEN and ten different variants for the two-level STRASSEN. These variants have different operand counts $W_{\{A,B,C\}}$, as shown in Figure 7.

Var#0 and Var#1: Instance ⓪ in (3), whose predicates are all `true` (non-zero), forms Var#0. That is, the instance in Var#0, with the operand counts $W_A = W_B = W_C = 2$, contains additional register allocation at line 3, additional `mops` at line 13, 15, and 24, as well as additional `flops` at line 20 and 21. Var#1 (with scalar $\gamma_1 = 0$) contains Instances ⑤ and ⑥. As a result, instances in Var#1, with the operand counts $W_A = W_B = 2$ and $W_C = 1$, do not perform extra post-processing on the output, hence with fewer `mops`.

Stage	Operation		`stream`
0	① $M_1 = (A_2 + A_3)B_0;$	$C_2 {+}{=}\ M_1; C_3 {-}{=}\ M_1;$	[0]
	④ $M_4 = (A_0 + A_1)B_3;$	$C_1 {+}{=}\ M_4; C_0 {-}{=}\ M_4;$	[1]
	⑤ $M_5 = (A_2 - A_0)(B_0 + B_1);$	$C_3 {+}{=}\ M_5;$	[0]
	⑥ $M_6 = (A_1 - A_3)(B_2 + B_3);$	$C_0 {+}{=}\ M_6;$	[1]
1	② $M_2 = A_0(B_1 - B_3);$	$C_1 {+}{=}\ M_2; C_3 {+}{=}\ M_2;$	[0]
	③ $M_3 = A_3(B_2 - B_0);$	$C_0 {+}{=}\ M_3; C_2 {+}{=}\ M_3;$	[1]
2	⓪ $M_0 = (A_0 + A_3)(B_0 + B_3);$	$C_0 {+}{=}\ M_0; C_3 {+}{=}\ M_0;$	[0]

Figure 8: Reordered operations based on (3) with multi-kernel streaming.

Both variants allocate registers $next1_A[m_R]$ and $next1_B[n_R]$, consuming the most registers out of the four variants.

Var#2 and Var#3: Instances ② and ③, whose predicate δ is `false`, form Var#2, with the operand counts $W_A = 1$ and $W_B = W_C = 2$. Because scalar $\delta = 0$, only registers $next1_A[m_R]$ will be allocated. Registers $next1_A[m_R]$ will be optimized out (through dead code elimination), since they will never be used. Similarly, Instances ① and ④ form Var#3, which has the operand counts $W_B = 1$ and $W_A = W_C = 2$ and only allocates registers $next1_B[n_R]$. These two variants have smaller register pressure, typically performing slightly better (with higher `FLOPS`) than Var#0 and Var#1 when the problem sizes are large. See Section 5 for a quantitative analysis on how these variants affect the performance.

3.3 Task Parallelism

A straightforward implementation of Strassen based on our specialized kernel (Section 3.1) invokes a sequence of GPU kernels sequentially (7 kernels for 1-level, 49 kernels for 2-level). This approach achieves intra-kernel parallelism across the thread blocks, warps, and threads, which is utilized in the GEMM implementation on a GPU. However, it is further possible to improve concurrency by exploiting more inter-kernel parallelism.

A careful look at (3) reveals that (i) the ordering of these operations can be arbitrary; (ii) the dependencies between the kernels for these operations only occur for the concurrent writes to different submatrices of C. That is, as long as race conditions are resolved, we can compute several instances in (3) simultaneously. Inter-kernel parallelism is especially important for small problem sizes when there is limited intra-kernel parallelism such that each kernel cannot saturate the workload on the GPU device and for multi-level STRASSEN when the partitioned block sizes are small. We next present three schemes to achieve this goal.

Streaming with dependencies: By invoking multiple independent kernels without write dependencies to different parts of C, we can achieve inter-kernel parallelism. To be specific, the seven instances in (3) can be rearranged into three synchronous stages (Stage 0–2) according to the dependency analysis, where kernels in the same stage can be executed asynchronously with two CUDA `streams`[5] (`stream[0]` and `stream[1]`).

In Figure 8, Stage 0 contains four instances. Instances ① and ④ can be executed concurrently with `stream[0]` and `stream[1]`. Instance ⑤ can be executed right after ① using `stream[0]` to avoid the possible race condition, and ⑥ can be executed using `stream[1]` in the same way. Instances ② and ③ are executed concurrently in Stage 2, and Stage 3 only contains Instance ⓪. Both streams must be synchronized at the end of each stage to enforce the order.

Element-wise atomic update: Although the first scheme works reasonably well for large problem sizes (where inter-kernel parallelism is less crucial), two streams do not expose enough parallelism for small and medium problem sizes (say $m = n = k \leq 6000$). Instead of resolving the race condition in the granularity of kernels, we exploit *out-of-order* parallelism at a finer granularity using atomic operations to resolve the possible concurrent write conflicts on matrix C. This is done by replacing the normal `Add` in the *Accumulator*

[5] CUDA programs can manage the concurrency across kernels through *streams* [25], each of which is a sequence of commands that execute in order. While the kernels launched within the same stream must be scheduled in sequential order, the commands from different streams may run concurrently out of order. To ensure every command in a particular stream has finished execution, `cudaDeviceSynchronize` can be used to enforce synchronization points.

with a global atomicAdd instruction. As a result, the 7 instances can all be executed concurrently with up to 7 CUDA streams.

Batching: With atomicAdd, 1-level Strassen launches 7 kernels concurrently, and 2-level Strassen may launch up to 49 kernels simultaneously. Although multiple streams can introduce more parallelism, the performance can easily be compromised by the kernel launching and context switch overhead, which is proportional to the number of streams and kernels. The overhead can even slow down the overall runtime when the problem size is small. As a result, we seek to launch the minimum number of kernels and streams by batching instances according to their variants. Instances in the same variant can be realized as a sequence (batch) of independent Strassen primitives (given the race condition on C is resolved by adtomicAdd).

To be specific, we use four streams to launch four GPU kernels concurrently. For example, the two instances in Var#1 are grouped as a batch of two, and the kernel is launched with 3D-grid, where the *z-dimension* equals the batch size. blockIdx.x and blockIdx.y are used to create the 2D-thread-block as usual to exploit parallelism within each Strassen instance. The additional blockIdx.z is used as an offset to exploit task-based parallelism between Strassen instances and access the proper pointers and scalars toward X, Y, V, W, D, E, δ, ϵ, γ_0, and γ_1.

3.4 Two-Level Strassen's Algorithm

Direct 2-level Strassen: Following [14], we can derive 49 instances (10 variants) from the general 2-level Strassen primitive that resembles (4) but with up to four submatrix operations in each operand. In the hierarchical view of Figure 5, we need to load four submatrices while packing the A and B *Tiles* from the **Device Level**. We also need to write the output back to four submatrices from the **Thread Block Level**. In Algorithm 1, we need to allocate extra register blocks $next2_A[m_R]$, $next2_B[n_R]$, $next3_A[m_R]$, and $next3_B[n_R]$ at line 3. Additional mops are introduced at line 13, 15, and 24. There are also additional flops introduced at line 20 and 21. As we can observe, although implementing a 2-level Strassen primitive can get rid of extra space requirement, the trade-off (regarding the current NVIDIA GPU architecture) is to increase the register pressure and the required memory bandwidth. As a result, the occupancy and floating point operation efficiency may be compromised. For a discussion on how this can be resolved in the future, see Section 5.4.

Hybrid 2-level Strassen: Alternatively, we combine the reference approach [20] with our specialized kernel to relieve the register pressure and the required memory bandwidth. The idea is to first apply the reference approach in [20], which requires $\mathcal{O}(mk/4 + kn/4 + mn/4)$ workspace. Then we apply our 1-level Strassen primitive to each of the seven submatrix multiplications. Together, we have a hybrid 2-level Strassen algorithm that consumes the same amount of workspace as [20] but ramps up much faster with smaller problem sizes. We empirically compare our hybrid approach with [20] in Section 4.

3.5 Handling the Fringes

Traditionally, for matrices with odd dimensions, we need to handle the remaining fringes before applying Strassen. There are some well-known approaches such as padding (i.e., adding rows or columns with zeros to get matrices of even dimensions) and peeling (i.e., deleting rows or columns to obtain even dimensioned matrices) [15, 33] followed by post-processing. In our approach, fringes can be internally handled by padding the *A Tile* and *B Tile* with zeros, and aligning the $m_C \times n_C$ *C Accumulator* along the fringes. This trick avoids the handling of the fringes with extra memory or computations because the packing and accumulation processes always occur for the high-performance implementation of gemm on GPUs, and we reuse the same buffers.

4 Experiment

We conduct three sets of experiments in Figure 9, providing an overview of our 1-level and 2-level Strassen. We discuss and analyze the performance of our algorithms through modeling in Section 5.

Setup: We perform our experiments on a Tesla V100 SXM2 accelerator which is connected to an Intel Xeon Gold 6132 Skylake server. The Operating System is CentOS Linux version 7.4.1708. The GNU compiler version for compiling the host code is 6.4.0. We use CUDA Toolkit 9.1 and compile the code with flags `-O3 -Xptxas -v -std=c++11 -gencode arch=compute_70,code=sm_70`. As presented in Section 2.2, the tested Tesla V100 SXM2 accelerator has a theoretical peak performance of 15.67 TFLOPS in single precision.

Measurement: We report the single precision floating point efficiency with three different configurations in Figure 9. We fix the ratio of m, n, and k dimension in the first configuration such that all matrices are square. In the second configuration, we fix $k = 4,096$ and vary m, n, resulting in tall-and-skinny matrix-multiplication (rank-k update). In the last configuration, we fix $m = n = 8,192$ and vary k, resulting in short-and-fat matrix-multiplication (panel dot-product).

To measure the execution time of GPU kernels running, we use CUDA events that have a resolution of approximately half a microsecond. We take *Effective* TFLOPS as the main metric to compare the performance of various implementations. To be specific,

$$\textit{Effective}\ \texttt{TFLOPS} = \frac{2 \cdot m \cdot n \cdot k}{\text{time (in seconds)}} \cdot 10^{-12}. \tag{6}$$

CUTLASS and our methods are tested with different strategies and block sizes to select the highest performing setup.

Result: In Figure 9, we report the single precision floating point efficiency of cuBLAS, CUTLASS, and various Strassen implementations on a V100 GPU. The 1-level and 2-level reference implementations [20] are linked with cuBLAS 9.1. For the 2-level hybrid implementation, we use reference implementation at the top level and our 1-level implementation at the bottom level.

By comparing the performance of various implementations, we make the following observations:

- For 1-level, our Strassen implementation outperforms CUTLASS and cuBLAS when the problem sizes $m = n = k$ are as small as 1,536. The reference implementation cannot get the comparable performance with our implementation until the problem sizes are larger than 10,000. For 2-level, our hybrid implementation outperforms the reference implementation.

- Our implementation has the same memory consumption as CUTLASS, while the 1-level reference implementation consumes much more memory. With V100 GPU (16 GB global memory), our 1-level Strassen can compute matrix multiplication for square problem sizes as large as 36,000, while the reference implementation runs out of memory after reaching 22,500.

- Our 1-level and hybrid 2-level Strassen implementations achieve the best performance over the entire spectrum of problem sizes compared to the reference implementations, with no or less additional memory consumption. Our hybrid 2-level implementation can get up to 1.22× (ideally 1.3×) speedup compared to CUTLASS and 1.19× speedup compared to cuBLAS when $m = n = k = 20,480$.

In summary, our 1-level Strassen algorithm can achieve practical speedup even for small (say $< 3,000$) and non-square matrices without using any extra workspace. As a result, our methods can easily benefit different matrix shapes and be applied to different applications such as matrix decomposition and tensor contraction. For large problem sizes ($> 9,000$), our hybrid 2-level Strassen algorithm can further provide speedup over our 1-level algorithm with additional $\mathcal{O}(mk/4 + kn/4 + mn/4)$ workspace.

5 Analysis

In this section, we analyze our performance results by deriving a performance model for gemm and different variants (Section 3.2) from Strassen. Performance modeling helps us select the right blocking parameters, predict the performance, and understand the computation and memory footprint of gemm and different Strassen implementations.

5.1 Notation and Assumptions

We summarize the notation in Figure 10 and assume the same three-level memory hierarchy as discussed in Section 2.1. For a thread block, the data movement through the memory hierarchy includes the following

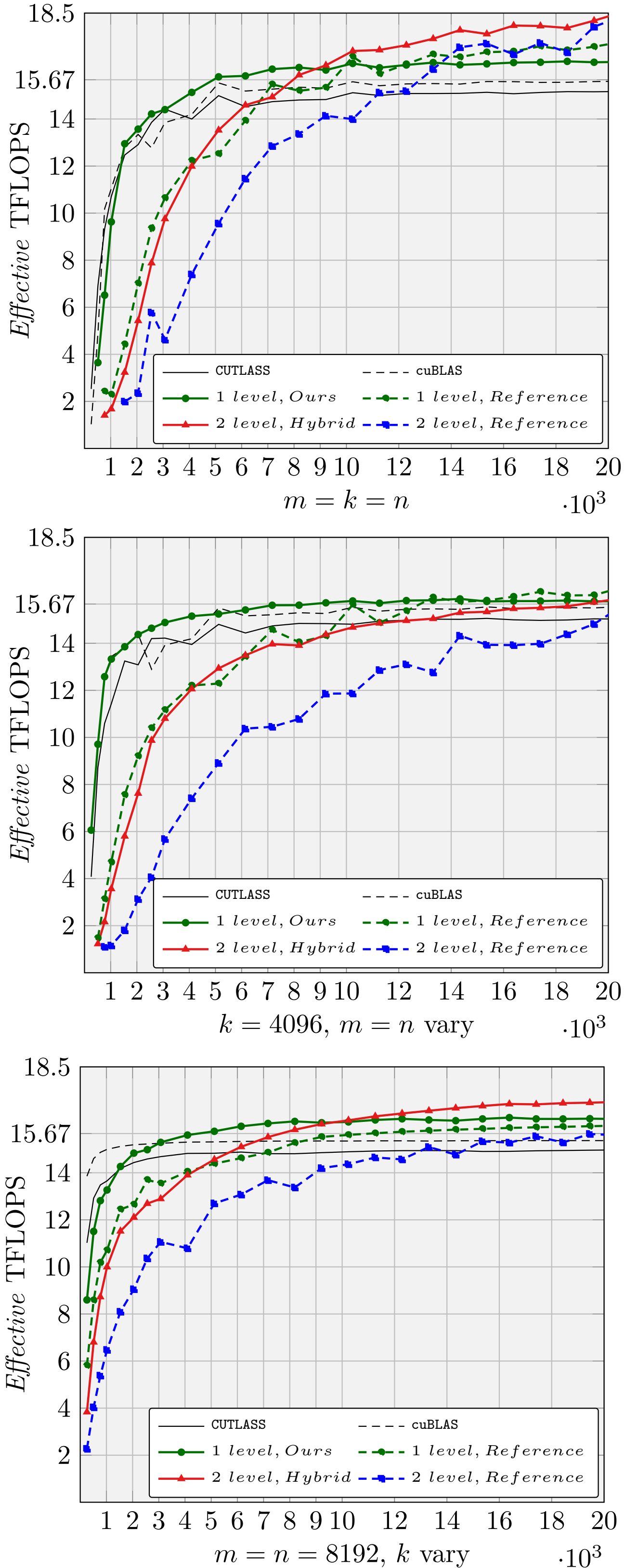

Figure 9: Performance of various Strassen implementations on V100 with single precision: the **x-axis** denotes the matrix size, and the **y-axis** denotes the floating point efficiency in `TFLOPS`. Our 1-level and hybrid 2-level implementations are built on `CUTLASS`, while the reference implementations are linked with `cuBLAS`.

Notation	Description	Value
$\tau_{\texttt{flop}}$	Arithmetic operation throughput	15.67 TFLOPS
$\tau_{\texttt{gmop}}$	Global memory bandwidth	1.08 TMOPS
$\tau_{\texttt{smop}}$	Shared memory bandwidth	15.30 TMOPS
T	Total execution time (in seconds)	
$t_x t_y$	Number of threads per thread block	$(m_S n_S)/(m_R n_R)$
$N^{\times}_{\texttt{flop}}$	Total `flops` for GEMM per thread block	$2 m_S n_S k$
$N^{+}_{\texttt{flop}}(A)$	Extra + operations for operand A	$(W_A - 1) m_S k$
$N^{+}_{\texttt{flop}}(B)$	Extra + operations for operand B	$(W_B - 1) n_S k$
$N^{+}_{\texttt{flop}}(C)$	Extra + operations for operand C	$(W_C - 1) m_S n_S$
$N_{\texttt{flop}}$	Total `flops` per thread block	Equation (10)
$T_{\texttt{flop}}$	Time for arithmetic operations	Equation (19)
$N_{\texttt{gmop}}(X)$	Global `mops` per block for operand X	Equations (7) (9)
$N_{\texttt{gmop}}$	Global memory operations per block	Equation (11)
$T_{\texttt{gmop}}$	Time for global memory operations	Equation (21)
$N_{\texttt{smop}}(X)$	Shared `mops` per block for operand X	Equations (7) (8)
$N_{\texttt{smop}}$	Shared operations per block	Equation (13)
$T_{\texttt{smop}}$	Time for shared memory operations	Equation (20)

Figure 10: Notation table for performance analysis.

primitives:
(i) loading the A and B *Tile* for k/k_S times from global memory to shared memory, which is further decomposed into two steps: prefetching from global memory to register files (line 12–15 in Algorithm 1) and storing back from register files to shared memory (line 20–21):

$$\begin{aligned} N_{\texttt{gmop}}(A_{\texttt{gr}}) = N_{\texttt{smop}}(A_{\texttt{rs}}) = m_S k_S (k/k_S), \\ N_{\texttt{gmop}}(B_{\texttt{gr}}) = N_{\texttt{smop}}(B_{\texttt{rs}}) = n_S k_S (k/k_S). \end{aligned} \tag{7}$$

(ii) loading the A and B *Fragment* from shared memory to register files (line 17–18):

$$\begin{aligned} N_{\texttt{smop}}(A_{\texttt{sr}}) = t_x t_y m_R k_S (k/k_S), \\ N_{\texttt{smop}}(B_{\texttt{sr}}) = t_x t_y n_R k_S (k/k_S). \end{aligned} \tag{8}$$

(iii) writing back the C *Accumulator* from register files to global memory (line 23):

$$N_{\texttt{gmop}}(C_{\texttt{rg}}) = m_S n_S. \tag{9}$$

The total number of arithmetic operations for one thread block, $N_{\texttt{flop}}$, can be decomposed into matrix multiplications $N^{\times}_{\texttt{flop}}$ and extra matrix additions

$$N_{\texttt{flop}} = N^{\times}_{\texttt{flop}} + N^{+}_{\texttt{flop}}(A) + N^{+}_{\texttt{flop}}(B) + N^{+}_{\texttt{flop}}(C). \tag{10}$$

Due to the prefetching pipeline, memory operations (handled by memory units) are overlapped with the arithmetic operations (handled by CUDA cores). We do not consider L1/L2 hardware cache effect, but we do take the read-only cache (texture memory) effect into account. We also do not consider the impacts of the task parallelism.

5.2 Blocking Parameter Selection

Similar to [32, 36], we select the blocking parameters for GEMM and different STRASSEN variants (Section 3.2) by analyzing the hardware constraints such as the maximum register number per thread and the memory bandwidth. Note that the following analysis mainly applies to large problem sizes when all SMs on V100 are fully utilized. We assume $\tau_{\texttt{flop}} = 15.67$ TFLOPS (Section 2.2), $\tau_{\texttt{gmop}} = 1.08$ TMOPS[6], $\tau_{\texttt{smop}} = 15.30$ TMOPS[7], $m_S = n_S$, and $m_R = n_R$ for square matrice cases. The bounds for the blocking sizes are loose.

Global memory bandwidth upper bound: Each thread block computes $N_{\texttt{flop}}$ arithmetic operations and reads

$$N_{\texttt{gmop}} = N_{\texttt{gmop}}(A_{\texttt{gr}})W_A + N_{\texttt{gmop}}(B_{\texttt{gr}})W_B + N_{\texttt{gmop}}(C_{\texttt{gr}})W_C \tag{11}$$

words. We can derive the bounds of m_S and n_S as

$$(N_{\texttt{flop}}/N_{\texttt{gmop}}) \geq \texttt{sizeof(float)}(\tau_{\texttt{flop}}/\tau_{\texttt{gmop}}). \tag{12}$$

It can be shown that $m_S = n_S \geq 58.2$, which results in the "Large" and "Huge" strategies for GEMM. For 1-level STRASSEN where the total reads may double (e.g., Var#0 and #1), we need to choose the "Huge" strategy where $m_S = n_S = 128$. For 2-level STRASSEN, the required block sizes can be up to four times large. As a result, no strategy is suitable.

Shared memory bandwidth upper bound: Similarly, each thread block reads and writes

$$N_{\texttt{smop}} = N_{\texttt{smop}}(A_{\texttt{sr}}) + N_{\texttt{smop}}(A_{\texttt{rs}}) + N_{\texttt{smop}}(B_{\texttt{sr}}) + N_{\texttt{smop}}(B_{\texttt{rs}}). \tag{13}$$

We can derive the bounds of block sizes m_R and n_R as

$$(N_{\texttt{flop}}/N_{\texttt{smop}}) \geq \texttt{sizeof(float)}(\tau_{\texttt{flop}}/\tau_{\texttt{smop}}). \tag{14}$$

As a result, we can get $m_R = n_R \geq 4.1$.

Register number per thread constraint: In Algorithm 1, each thread requires $m_R \times n_R$ registers for the accumulator, $(W_A m_R + W_B n_R)$ for fetching and prefetching operands A and B, and $2(m_R + n_R)$ for double buffering operands between shared memory and register files.[8] Since the maximum registers per thread is 255, m_R and n_R are bounded by

$$m_R n_R + (2 + W_A)m_R + (2 + W_B)n_R < 255. \tag{15}$$

We can get $m_R = n_R < 12$.

Shared memory size per SM constraint: Each thread block keeps the $\{A,\ B\}$ *Tile* in the shared memory, which requires

$$\texttt{sizeof(float)}(m_S k_S + n_S k_S) < 96K, \tag{16}$$

since the shared memory capacity per SM is 96 KB.

Global memory prefetching precondition: Each thread prefetches one subcolumn of A with height m_R (line 12) and one subrow of B with width n_R (line 14), all $t_x \times t_y$ threads in one thread block need to store back to the $m_S k_S$ *A Tile* (line 20) and the $n_S k_S$ *B Tile* (line 21), so it requires

$$m_R t_x t_y \geq m_S k_S, n_R t_x t_y \geq n_S k_S. \tag{17}$$

We can therefore get $k_S \leq m_S/m_R$, $k_S \leq n_S/n_R$.

Basically, the Huge strategy in CUTLASS (Section 2.3) meets the bound requirement to maximize the performance for both GEMM and different variants from 1-level STRASSEN (Var#0-#3) on large problem sizes.

[6] Due to the read-only cache (texture memory) effect, the global memory bandwidth is enhanced by a factor of 20%, i.e., 900 (GB/s) × (1+20%).

[7] 80 (# SM) × 32 (# banks/SM) × 4 (# bank width: Bytes) × 1530 MHz [16].

[8] At least $W_A + W_B + 5$ additional registers are needed: $W_A + W_B$ registers to track A, B in the global memory during prefetching (line 12–15); 1 register to store the loop end condition; 2 registers to track A, B in the shared memory when prefetching (line 17–18); 2 registers to track A, B in the shared memory for storing back (line 20–21).

5.3 Performance Prediction

The total execution time T can be estimated as the maximum of the time of arithmetic operations $T_{\mathtt{flop}}$, the shared memory operations $T_{\mathtt{smop}}$, and the global memory operations $T_{\mathtt{gmop}}$. That is, $T = \max(T_{\mathtt{flop}}, T_{\mathtt{gmop}}, T_{\mathtt{smop}})$.
Arithmetic operations: We assume that the computation power of a GPU is split evenly among all active thread blocks, i.e., each active thread block can get a portion of the peak throughput $\tau_{\mathtt{flop}}$ of the whole GPU device: $\tau_{\mathtt{flop}}/\#blocks$. Here #blocks is the maximum active thread blocks on one V100 device, which is computed by

$$\#blocks = \#SM \times \#max\_active\_blocks\_per\_SM^{9}. \tag{18}$$

As a result, for $\lceil \frac{m}{m_S} \rceil \lceil \frac{n}{n_S} \rceil$ submatrix blocks, the total arithmetic operation time is

$$T_{\mathtt{flop}} = \left\lceil \frac{\lceil \frac{m}{m_S} \rceil \lceil \frac{n}{n_S} \rceil}{\#blocks} \right\rceil \left(\frac{\#blocks \times N_{\mathtt{flop}}}{\tau_{\mathtt{flop}}} \right) \tag{19}$$

for $\#blocks$ active thread blocks.
Shared memory operations: Similarly, we assume that the bandwidth of shared memory is allocated evenly to each active thread block. Given that the number of shared memory operations per thread block in (13), the total time spent on shared memory operations is

$$T_{\mathtt{smop}} = \left\lceil \frac{\lceil \frac{m}{m_S} \rceil \lceil \frac{n}{n_S} \rceil}{\#blocks} \right\rceil \left(\frac{\mathtt{sizeof(float)}\#blocks \times N_{\mathtt{smop}}}{\tau_{smop}} \right). \tag{20}$$

Global memory operations: The global memory is accessible by all threads on all SMs and resides on the device level, so the bandwidth is not necessarily divided evenly by all thread blocks.[10] Given the number of global memory operations per thread block in (11), the total time spent on global memory operations is

$$T_{\mathtt{gmop}} = \lceil \frac{m}{m_S} \rceil \lceil \frac{n}{n_S} \rceil \left(\frac{\mathtt{sizeof(float)} N_{\mathtt{gmop}}}{\tau_{\mathtt{gmop}}} \right). \tag{21}$$

We can predict the run time performance of various implementations, based on this performance model. In Figure 11, we present the modeled and actual performance of GEMM and direct 1/2-level STRASSEN (Sections 3.1 and 3.4) for square matrices with Huge and Small strategies of block sizes (Figure 4). The direct 1- and 2-level STRASSEN are implemented using 7/49 instances of different variants sequentially, without inter-kernel task parallelism (Section 3.3).

5.4 Discussion and Analysis

Impacts of the variants in STRASSEN: From our model, the performance differences between the variants (Section 3.2) are determined by the operand counts $W_{\{A,B,C\}}$, which mainly affects the number of global memory operations (12) and $T_{\mathtt{gmop}}$, the total number of arithmetic operations $N_{\mathtt{flop}}$, and the register number (15). For example, comparing Var#0 in 1-level STRASSEN with GEMM, we can find that the global memory operation number doubles, and the required register number increases by $m_R + n_R$.
Limitations and possible solutions: Our 2-level STRASSEN primitives may increase operands count W_A, W_B, and W_C up to four times. These primitives may require up to 160 registers per thread by (15), and up to 1,900 GB/s global and texture memory throughput by (12). Regarding the current architecture, memory operations cannot be fully overlapped with the computations and registers must be spilled to maintain two active thread block per SM (or just maintain one active thread block). These two limitation factors suggest possible hardware improvements on future generation GPUs to make the 2-level primitives practical.

[9] $\#max\_active\_blocks\_per\_SM$ denotes the number of the maximum active thread blocks per SM, which can be returned from function `cudaOccupancyMaxActiveBlocksPerMultiprocessor`, or calculated with the CUDA Occupancy Calculator provided by NVIDIA [26]. For Huge, GEMM and all variants: 2; For Small, GEMM: 24, 1-level Var#0: 20, 2-level Var#0: 18.

[10] On the hardware layer, the HBM2 memory is connected to the chips through eight memory controllers in four memory stacks [28], not coupled with individual SMs.

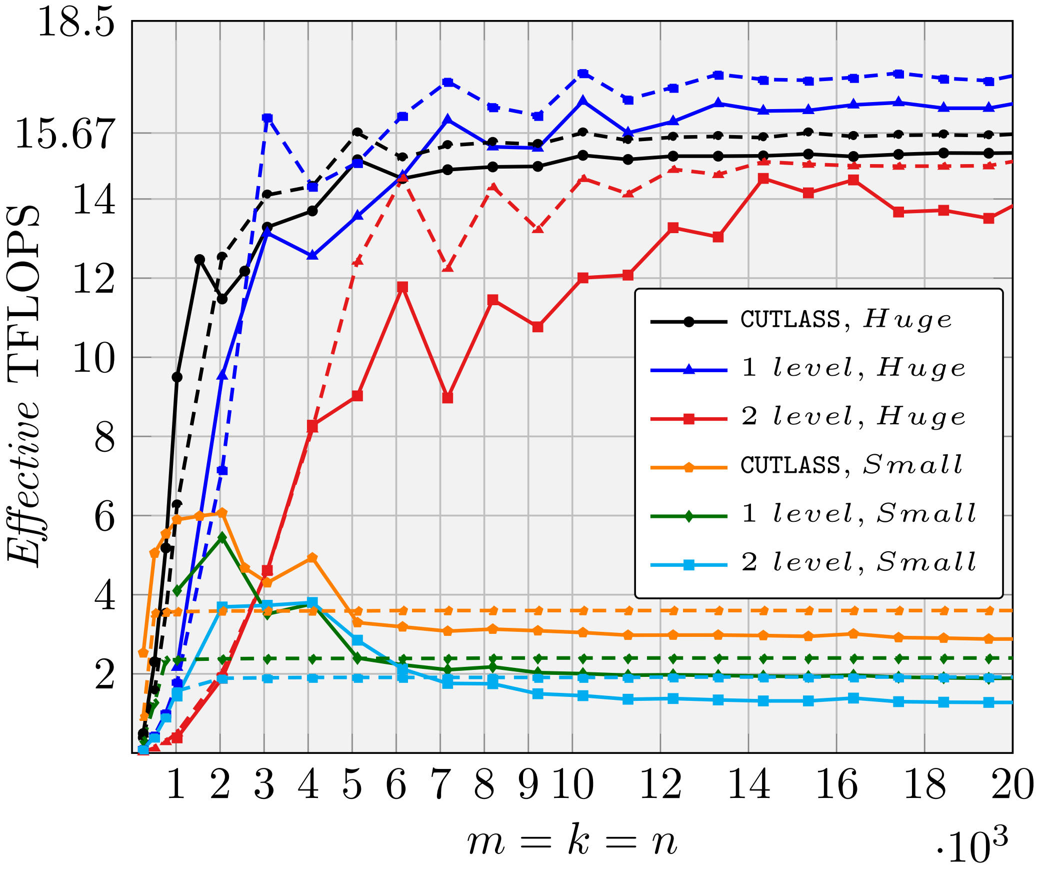

Figure 11: Actual (solid line) and modeled (dashed line) performance of `CUTLASS` and Strassen with Small and Huge strategies of block sizes.

Extra registers $next1_A$ and $next1_B$ in Algorithm 1 are used to prefetch extra operands at line 12–15, which are handled solely by the memory units thus overlapped with rank-k_S update during line 17–19. For 2-level Strassen, the extra registers required for prefetching will exceed the constraint. Moving arithmetic operations at line 20–21 to line 12–15 can reduce the register requirement by reusing $next1_A$ and $next1_B$ but result in CUDA cores waiting for the memory access, thus decrease the number of overlapped memory operations. Given that the 2-level Strassen primitives already require much higher memory bandwidth, it is not practical to trade overlapped memory operations with more registers.

To alleviate the register pressure and memory traffic, our Strassen primitives are good examples that could benefit from Processing-In-Memory (PIM) [3, 1, 22]. With extended memory instructions that directly compute the arithmetic operations at line 20–21 during the fetching process at line 12–15, it is possible to remove all extra registers for prefetching. The computation is done in-transit of the loading process, which may also relieve the memory traffic in the memory hierarchy and reduce the required memory throughput.
Cache effects: For the Small strategy, the actual performance is better than the modeled performance during the ramp-up stage. This shows the L1/L2 cache effects as there are two performance "falling edges" for the actual performance, which are not captured by our performance model.

6 Conclusion

We have presented a practical implementation of Strassen's algorithm on GPUs, which outperforms the state-of-the-art implementation on small problem sizes and consumes no additional memory compared to

GEMM. By developing a specialized kernel, we utilized the memory and thread hierarchies on GPUs. By reusing the shared memory to store the temporary matrix sum during the packing process and the register files to hold the temporary matrix product during the accumulation process, we avoided the extra workspace requirement and reduced the additional memory movement. Besides the intra-kernel parallelism across the thread blocks, warps, and threads similar to GEMM implementation on GPUs, we also exploited the inter-kernel parallelism and batched parallelism and overlapped the bandwidth limited operations with the computation bound operations. We demonstrated performance benefits for small and non-square matrices on a most recent Volta GPU, and verified the performance results by building an accurate performance model to choose the appropriate block sizes and predict the run time performance. Together, we achieved both less memory and more parallelism with our customized kernels. In the future, we will extend this work to other applications on GPUs, such as fast matrix multiplication algorithms [4, 13], high-dimensional tensor contractions [23], and convolution neural network [18, 29].

Acknowledgments

This work was sponsored in part by the National Science Foundation under grant number CCF-1714091 and the 2016 Peter O'Donnell Distringuished Research Award. This work was also made possible, in part, through HPC time donated by Microway, Inc. We gratefully acknowledge Microway for providing access to their GPU-accelerated compute cluster. Access to the Maverick2 supercomputers administered by TACC is gratefully acknowledged. We thank the rest of the SHPC team (`http://shpc.ices.utexas.edu`) for their supports.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 336–348. IEEE, 2015.

[2] Grey Ballard, Austin R Benson, Alex Druinsky, Benjamin Lipshitz, and Oded Schwartz. Improving the numerical stability of fast matrix multiplication. *SIAM Journal on Matrix Analysis and Applications*, 37(4):1382–1418, 2016.

[3] Janine C Bennett, Hasan Abbasi, Peer-Timo Bremer, Ray Grout, Attila Gyulassy, Tong Jin, Scott Klasky, Hemanth Kolla, Manish Parashar, Valerio Pascucci, et al. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 49. IEEE Computer Society Press, 2012.

[4] Austin R. Benson and Grey Ballard. A framework for practical parallel fast matrix multiplication. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*, pages 42–53. ACM, 2015.

[5] CUTLASS: CUDA templates for linear algebra subroutines (v0.1.0). GitHub Repository, 2018. `https://github.com/NVIDIA/cutlass/releases/tag/v0.1.0`.

[6] Paolo D'Alberto, Marco Bodrato, and Alexandru Nicolau. Exploiting parallelism in matrix-computation kernels for symmetric multiprocessor systems: Matrix-multiplication and matrix-addition algorithm optimizations by software pipelining and threads allocation. *ACM Trans. Math. Softw.*, 38(1):2:1–2:30, December 2011.

[7] James Demmel, Ioana Dumitriu, Olga Holtz, and Robert Kleinberg. Fast matrix multiplication is stable. *Numerische Mathematik*, 106(2):199–224, 2007.

[8] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[9] Luke Durant, Olivier Giroux, Mark Harris, and Nick Stam. Inside Volta: The world's most advanced data center GPU. NVIDIA Developer Blog, 2017. `https://devblogs.nvidia.com/inside-volta`.

[10] Scott Gray. Nervanagpu. Available Online, 2017. `https://github.com/NervanaSystems/maxas/wiki/SGEMM`.

[11] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, Philadelphia, PA, USA, second edition, 2002.

[12] Jianyu Huang, Devin A. Matthews, and Robert A. van de Geijn. Strassen's algorithm for tensor contraction. *SIAM Journal on Scientific Computing*, 40(3):C305–C326, 2018.

[13] Jianyu Huang, Leslie Rice, Devin A. Matthews, and Robert A. van de Geijn. Generating families of practical fast matrix multiplication algorithms. In *31th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2017)*, pages 656–667, May 2017.

[14] Jianyu Huang, Tyler M. Smith, Greg M. Henry, and Robert A. van de Geijn. Strassen's algorithm reloaded. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 16)*, pages 59:1–59:12. IEEE Press, 2016.

[15] Steven Huss-Lederman, Elaine M. Jacobson, Anna Tsao, Thomas Turnbull, and Jeremy R. Johnson. Implementation of Strassen's algorithm for matrix multiplication. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, SC 96, Washington, DC, USA, 1996. IEEE.

[16] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[17] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. CUTLASS: Fast linear algebra in CUDA C++. NVIDIA Developer Blog, Dec 2017. `https://devblogs.nvidia.com/cutlass-linear-algebra-cuda`.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[19] Junjie Lai and Andre Seznec. 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)*, CGO 13, pages 1–10, Washington, DC, USA, 2013. IEEE Computer Society.

[20] Pai-Wei Lai, Humayun Arafat, Venmugil Elango, and P. Sadayappan. Accelerating Strassen-Winograd's matrix multiplication algorithm on GPUs. In *20th Annual International Conference on High Performance Computing*, pages 139–148, Dec 2013.

[21] Junjie Li, Sanjay Ranka, and Sartaj Sahni. Strassen's matrix multiplication on GPUs. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 157–164, Dec 2011.

[22] Gabriel H Loh, Nuwan Jayasena, M Oskin, Mark Nutter, David Roberts, Mitesh Meswani, Dong Ping Zhang, and Mike Ignatowski. A processing in memory taxonomy and a case for studying fixed-function pim. In *Workshop on Near-Data Processing (WoNDP)*, 2013.

[23] Devin A. Matthews. High-performance tensor contraction without transposition. *SIAM Journal on Scientific Computing*, 40(1):C1–C24, 2018.

[24] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved magma gemm for fermi graphics processing units. *The International Journal of High Performance Computing Applications*, 24(4):511–515, 2010.

[25] NVIDIA. CUDA C programming guide. Available Online, 2018. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[26] NVIDIA. CUDA occupancy calculator. Available Online, 2018. https://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.

[27] NVIDIA. Nvidia cuBLAS. Available Online, 2018. https://developer.nvidia.com/cublas.

[28] NVIDIA. Tuning CUDA applications for volta. Available Online, 2018. https://docs.nvidia.com/cuda/volta-tuning-guide/index.html.

[29] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[30] Paul Springer and Paolo Bientinesi. Design of a high-performance GEMM-like tensor-tensor multiplication. *ACM Trans. Math. Softw.*, 44(3):28:1–28:29, January 2018.

[31] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, August 1969.

[32] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. Fast implementation of dgemm on fermi gpu. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 11, pages 35:1–35:11, New York, NY, USA, 2011. ACM.

[33] Mithuna Thottethodi, Siddhartha Chatterjee, and Alvin R. Lebeck. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC 98)*, pages 1–14. IEEE, 1998.

[34] Vasily Volkov and James W Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC 08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–11, Nov 2008.

[35] Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the K-Nearest Neighbors kernel on x86 architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 15, pages 7:1–7:12, New York, NY, USA, 2015. ACM.

[36] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. Understanding the gpu microarchitecture to achieve bare-metal performance tuning. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 17, pages 31–43, New York, NY, USA, 2017. ACM.