A Message-Driven, Multi-GPU Parallel Sparse Triangular Solver *

Nan Ding† Yang Liu ‡ Samuel Williams† Xiaoye S. Li‡

Abstract

Sparse triangular solve is used in conjunction with Sparse LU for solving sparse linear systems, either as a direct solver or as a preconditioner. As GPUs have become a first-class compute citizen, designing an efficient and scalable SpTRSV on multi-GPU HPC systems is imperative. In this paper, we leverage the advantage of GPU-initiated data transfers of NVSHMEM to implement and evaluate a Multi-GPU SpTRSV. We create a novel producer-consumer paradigm to manage the computation and communication in SpTRSV and implement it using two CUDA streams. Our multi-GPU SpTRSV implementation using CUDA streams achieves a 3.7× speedup when using twelve GPUs (two nodes) relative to our implementation on a single GPU, and up to 6.1× compared to *cusparse_csrsv2()* over the range of one to eighteen GPUs. To further explain the observed performance and explore the key features of matrices to estimate the potential performance benefits when using multi-GPU, we extend the critical path model of SpTRSV to GPUs. We demonstrate the ability of our performance model to understand various aspects of performance and performance bottlenecks on multi-GPU and motivate code optimizations.

1 Introduction

Over the last decade, accelerated computing architectures have become more and more popular in modern HPC systems. A total of 147 systems on the 2020 TOP500 list are using accelerators [1], of which, 110 systems use NVIDIA Volta chips [2]. Concurrently, sparse triangular solve (SpTRSV) is considered an indispensable task in a wide range of applications from numerical simulation [3] to machine learning [4]. Designing an efficient and scalable sparse triangular solver (SpTRSV) on modern multi-GPU HPC systems is imperative but challenging. In recent years, substantial efforts have focused on single GPU SpTRSV [5, 6]. However, with more scientific insights derived from computation, the demand for ever finer-resolution problems calls for SpTRSV to exploit ever larger scales of parallelism. Unfortunately, given the slow pace in HBM memory capacity scaling, one cannot guarantee the problem can always fit into a single GPU's memory.

In this paper, we implement and evaluate a multi-GPU SpTRSV. We leverage NVSHMEM [7] to perform direct GPU-GPU communication. NVSHMEM is a parallel programming interface based on OpenSHMEM [8] that provides efficient and scalable (one-sided) communication for NVIDIA GPU clusters. The advantages of using NVSHMEM is that it uses GPU-initiated data transfers. This allows users to perform both computations and communications in one CUDA kernel instead of transferring data between the CPU and the GPU. Unfortunately, NVSHMEM also has a major limitation. It limits the number of thread blocks that can be concurrently scheduled on one V100 GPU to 80 to avoid potential deadlocks when using point-to-point synchronization in the CUDA kernel. Nominally, such a limitation would significantly restrict SpTRSV concurrency.

To overcome the concurrency limitation of NVSHMEM, we propose a coupled producer-consumer parallelism using CUDA streams. CUDA streams are often used to overlap computation and communication for the simple producer-consumer style of parallelism such as stencils where one stream performs all computations and the other stream handles communication [9]. Although a simple CUDA stream synchronize might suffice for stencils, SpTRSV has a more complex producer-consumer relationship — the producer (sender) and the consumer (receiver) can swap roles in turn to dispatch new work (message). We use two CUDA streams to handle this complex coupled producer-consumer parallelism. The advantages of using CUDA streams include not only the mitigation of the NVSHMEM concurrency

*This research is supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) programs under Contract No. DE-AC02-05CH11231 at Lawrence Berkeley National Laboratory. This research used resources of the the Oak Ridge Leadership Facility which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. We thank Akhil Langer from NVIDIA Corporation for his willingness to answer our myriad of questions on NVSHMEM.

†Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA, (nanding|swwilliams@lbl.gov)

‡Scalable Solvers Group, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA, (liuyangzhuan|xsli@lbl.gov)

limitation, but also enabling the of overlap of communication and computation.

Aligned with the advances in hardware and communication paradigms, performance modeling of SpTRSV is critical to assess potential performance gains in terms of machine capability. Modeling SpTRSV depends heavily on the structure of a given matrix and the underlying architecture. Whereas the Roofline model [10, 11] can effectively bound performance and identify bottlenecks for well-structured, load-balanced codes, it can only provide a very loose bound on performance for codes like SpTRSV. To that end, a SpTRSV performance model is proposed in work [12] for pure MPI implementations on CPUs. The model is based on the critical path analysis which follows the task dependency graph of the sparse matrix using the well-known level-set method [13, 14] with a breadth-first search [15]. Process decomposition is also considered into the critical path analysis. The computations and communications in each MPI process are serialized. Therefore, it lacks concurrency in each process (GPU), including concurrent messaging and computation from the thread blocks in one GPU (corresponding to one MPI process in [12]). Ultimately, the contributions in this paper include: **(1)** Develop a method of coupled producer-consumer parallelism using CUDA streams which overcomes the concurrency limitation of NVSHMEM, and overlap the communications and computations, **(2)** Implement a multi-GPU (both intra- and inter-node) SpTRSV using coupled CUDA streams which achieves a 3.7× speedup when using twelve GPUs (two nodes) compared to the single GPU implementation on Summit, and **(3)** Extend our SpTRSV performance model for GPUs that enables insights into various aspects of performance and performance bottlenecks on multiple GPUs.

2 Distributed-memory Parallel Sparse Triangular Solve

SpTRSV computes a solution vector x for a $n \times n$ linear system $Lx = b$ [1], where L is a lower triangular matrix, and b is a $n \times k$ right-hand side (RHS) matrix or vector ($k = 1$). For a sparse matrix L, the computation of x_i needs some or all of the previous solution rows x_j, $j < i$, depending on the sparsity pattern of the i^{th} row of L. This computation dependency can be precisely expressed by a DAG. We use a supernodal DAG [16] formulation. For a lower triangular matrix L, a supernode is a set of consecutive columns of L with the triangular block just below the diagonal being full, and the same nonzero structure below the triangular block. After a supernode partition is obtained along the columns of L, we apply the same partition row-wise to obtain a 2D block partitioning. The nonzero block pattern defines the supernodal DAG. We assume $b(K)$ and $x(K)$ represent the subvector associated with supernode K. $L(I, K)$ denotes the nonzero submatrix corresponding to supdernodes I and K. Thus, the solution of subvector $x(K)$ can be computed as Eq. (2.1).

$$x(K) = L(K,K)^{-1}\Big(b(K) - \sum_{I=1}^{K-1} L(K,I) \cdot x(I)\Big) \tag{2.1}$$

The distributed-memory SpTRSV [12] partitions the matrix L among multiple processes using a 2D block cyclic layout. Each process is in charge of a subset of solution subvectors $x(K)$. The solution of these subvectors and partial summation results require communication. Figure 1 describes the data flow of a sparse triangular solve using a 2×2 process decomposition. Processes assigned to the diagonal blocks, called *diagonal processes*, compute the corresponding blocks of x. In the process decomposition of Figure 1, processes {0, 4, 8} are the diagonal processes. Within one block column, the process owning $x(I)$ sends $x(I)$ to the process of $L(K, I)$ as No. ① in Figure 1. After receiving the required $x(I)$ subvector, each process computes its local summation. Within one row, the local sums are sent to the diagonal process which will perform the inversion (No. ② in Figure 1). In this case, process 0 is the producer, and process 6 is the consumer in the first block column. However, process 6 reverts to being the producer in the fifth block row. An asynchronous binary tree [17] is used to perform the column broadcast and row reduction, while a one-sided `MPI_Put` is used to reduce the communication latency. Each message contains the data and a checksum payload. The checksum payload is used by receivers to check completion. The binary tree is built in the setup phase that is executed once for multiple solves. A broadcast tree per supernode column K is built for the processes participating in the column broadcast. Similarly, one reduction tree is built per supernode row I within the processes participating in the row reduction. Note that each process in a tree need only keep track of its parent and children.

3 Benefits and Challenges of NVSHMEM

NVSHMEM is a parallel programming interface based on OpenSHMEM that provides efficient and scalable communication (one-sided) for NVIDIA GPU clusters. The advantages of using NVSHMEM for multi-GPU programming is that it uses GPU-initiated data

[1] We use lower triangular matrix to formulate the problem in the paper. Note that the proposed methodology can be easily ported to solve an upper triangular system $Ux = b$.

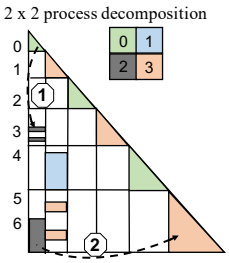


Figure 1: Data flow and process decomposition of the distributed-memory SpTRSV. The dashed arrows represent the data flow. The numbers represent two communication behaviors: ① block column broadcast, and ② block row reduction.

transfers. This feature allows programmers to execute both computations and communications in one CUDA kernel in lieu of initiating communication from the CPU.

Some numerical methods have a relatively simple communication pattern, such as stencils, which adhere to the Bulk Synchronous Parallel (BSP) model [18]. Inter-processor communications follow the discipline of strict barrier synchronization. As such, communication models like MPI and its CUDA-aware variant [19] can satisfy the requirements of those applications. Conversely, DAG-like computations, like SpTRSV, have a more complex communication pattern. Point-to-point communications can happen at any time between any two processes (depending on the sparsity pattern and the process decomposition) with no strict barrier synchronization. Therefore, the GPU-initiated communication nature of NVSHMEM provides a big advantage over other communication paradigms.

One challenge when using one-sided communication is how to notify receivers that the data has completely arrived. NVSHMEM provides signaling operations and point-to-point synchronization operations. These operations relieve users of the burden of implementing their own data synchronization. However, it also brings one limitation. When using synchronizations in the CUDA kernel, the number of thread blocks that can be launched on one V100 GPU is limited to 80 (the number of SMs) to avoid potential deadlocks. This is an inherent limitation of the NVSHMEM+CUDA+NVIDIA GPU environment, and can significantly restrict the concurrency in SpTRSV.

To overcome the limitation of 80 thread blocks, we propose a coupled producer-consumer parallelism using CUDA streams. We use two streams to execute two kernels concurrently. One kernel, named `WAIT` in *stream[0]*, handles NVSHMEM point-to-point synchronizations. It is launched using `nvshmemx_collective_launch()` with a number of thread blocks less than 80 (to ensure the GPU is not fully occupied). The other kernel, named `SOLVE` in *stream[1]*, is responsible for computation and sending data/notification. It is launched after the `WAIT` kernel as a normal CUDA kernel thereby maximizing concurrency. We use a bit scalar ($flag_w$, Algorithm 1 line 4) to control the launch order of the two kernels. The `WAIT` kernel is launched first, and sets the scalar to *True*. The `SOLVE` kernel is not launched until $flag_w == True$.

4 Multi-GPU SpTRSV using CUDA Streams.

Algorithm 1 details our design for our multi-GPU SpTRSV, and the variables are listed in Table 1. We bind one process to one GPU so that each GPU (corresponding to a process in Section 2) is in charge of a subset of solution subvectors $x(K)$.

Let us assume that a lower triangular matrix $L(n, n)$ has N supernodes in total, N_g supernode columns per GPU, and N_r supernode rows per GPU. We launch $N_g + 2$ thread blocks per GPU for the matrix L, where two thread blocks are for the `WAIT` kernel, and N_g thread blocks are used by the `SOLVE` kernel. The `SOLVE` kernel uses those N_g thread blocks to perform the requisite TRSV (Triangular Solve Matrix-Vector, diagonal blocks) and GEMV (General Matrix-Vector Multiplications, off-diagonal blocks) computations, broadcasts x subvector, notifies the consumers.

We perform row reductions in both kernels whenever the data dependency ($fmod(I) == 0$, line 21 and 49) is met. A counter $fmod(I)$ is computed per supernode row I to record the number of local inner-products and non-local messages for their contribution to $lsum(I)$. The number of local updates equals the number of blocks in row I one process owns, while the number of non-local updates is always no more than two (two children in the binary reduction tree).

In the pre-processing phase, we compute two masks M_c and M_r (line 2) for every GPU (process). A mask is a bit vector encoding a bit for each block column (M_c, size of N_g) or two bits of each block row (M_r, size of $2{\cdot}N_r$). According to the communication binary tree, one parent broadcast x subvector to two children at most. That is, one thread block waits for at most one message in each block column broadcast. In each row reduction, two (or one) children send its local summation ($lsum$) to their parent. Each thread block waits for at most two messages in a row reduction. The mask of a block column i, $M_c[i]$ (or row j, $M_r[j * 2]$ and $Mr[j * 2 + 1]$) represents whether block column i (or row j) needs communication or not. Thus, columns (or rows) that do not expect messages are masked. Each thread in the `WAIT` kernel has its own entry of the two masks.

Algorithm 1 Multi-GPU SpTRSV using two streams (variable definitions are listed in Table 1)

```
1: procedure PRE-PROCESSING(on CPU)
2:     Compute M_c, M_r for each GPU
3:     NVSHMEM launch WAIT, dimGrid(2), dimBlock(maxTH), stream[0]
4:     flag_w=0                ▷ the bit scalar to control the launch order
5:     while(flag_w!=1);                                  ▷ spin wait
6:     CUDA launch SOLVE, dimGrid(N_g), dimBlock(16x16), stream[1]
7: end procedure

8: procedure SOLVE(on GPU, stream[1])
9:     K = bid                 ▷ one thread block handles one block column
10:    if I am the diagonal process in charge of K then
11:        while(fmod(K)!=0);          ▷ spin wait, called by thread 0
12:        x(K)+ = lsum(K)
13:        x(K) = L(K,K)^{-1} · x(K)
                                        ▷ parallelize TRSV over threads
14:    else
15:        while(flag_x[K]!=1);        ▷ spin wait, called by thread 0
16:        NVSHMEM SEND ready_x(K) to my children's ready_x buffer
                                                  ▷ called by all threads
17:        for each L(I,K)! = 0, I > K do
                                  ▷ parallelize I and GEMV over threads
18:            lsum(I) = lsum(I) + L(I,K) · ready_x(K)
19:            fmod(I) = fmod(I) − 1
20:            if fmod(I) == 0 then
21:                NVSHMEM SEND ready_lsum(I) to my parent's ready_lsum buffer
                                                  ▷ called by one thread
22:            end if
23:        end for
24:    end if
25: end procedure

26: procedure WAIT(on GPU,stream[0])
27:    flag_w=1                 ▷ the bit scalar to control the launch order
28:    if bid==0 then                  ▷ handle block column broadcast
29:        while expecting more tasks do
30:            idx=nvshmem_int_wait_until_any(flag_x,M_c)
                                        ▷ M_c is distributed across threads
31:            M_c[idx]=1       ▷ message arrived in block column idx
32:        end while
33:    end if

34:    if bid==1 then                  ▷ handle block row reduction
35:        while expecting more tasks do
36:            idxr=nvshmem_int_wait_until_any(flag_lsum,M_r)
                                        ▷ M_r is distributed across threads
37:            M_r[idxr]=1     ▷ message arrived in block row idxr/2
38:            if (cnt[I]! = flag_lsum[I * 2] + flag_lsum[I * 2 + 1] then
                        ▷ cnt[I]: number of required messages in row I
          ▷ cnt[I]: pre-compute when building communication trees
39:                if flag_lsum[I * 2] == 1 then
40:                    lsum(I)+ = ready_lsum(I)
41:                    fmod(I) = fmod(I) − 1
42:                end if
43:                if flag_lsum[I * 2 + 1] == 1 then
44:                    lsum(I)+ = ready_lsum(I * 2)
45:                    fmod(I) = fmod(I) − 1
46:                end if
47:            end if
48:            if fmod(I) == 0 then
49:                NVSHMEM SEND ready_lsum(I) to my parent's ready_lsum buffer
                                                  ▷ called by one thread
50:            end if
51:        end while
52:    end if
53: end procedure
```

The `SOLVE` kernel in *stream[1]* (Algorithm 1 lines 8-25) performs computations (TRSV/GEMV), sends data and notification (lines 21 and 16) to receivers. As we assign one thread block to execute each supernode column, the number of thread blocks launched equals the number of local supernodes N_g on that GPU. Through empirical tuning, we use 256 threads for each thread block. Each GPU (process) performs TRSV first if it is a diagonal process (lines 10-14). Otherwise, it will spin wait until the message arrives ($flag\_x[K] = 1$: message arrived in block column K, line 15). Once it has received the message, that thread block immediately sends the x subvector to its children in the binary broadcast tree, and then performs GEMV computation, and sends $lsum$ if the counter $fmod(I)$ becomes zero. NVSHMEM SEND (line 16 and 21) is a set of NVSHMEM operations as Table 1 shows. The senders participating in column broadcast use three operations to send one message: ① `nvshmem_double_put_nbi_block()` called by thread blocks is used to send the data (x subvectors) to NVSHMEM buffer $ready\_x$. ② `nvshmemx_int_signal()` called by one thread is used to send the notification to NVSHMEM buffer $flag\_x$. ③ `nvshmem_fence()` is used between ① and ② to ensure that the notification is sent after the data. Senders in row reduction follow the same manner. The only difference is that they use `nvshmem_double_put_nbi()` called by one thread to send the $lsum$. Correspondingly, the point-to-point synchronization *nvshmemx_wait_until_any* in `WAIT` kernel guarantees that the message receive order preserves the message send order.

Table 1: Algorithm 1 References

	Descriptions		
N_g	number of thread blocks per GPU	`SOLVE` kernel	
N_r	number of block rows per GPU		
mz	maximum size of an individual message	equals to maximum supernode size	
bid	thread block id	1D grid	
$maxTH$	number of threads per block in `WAIT` kernel	1D, cudaOccupancyMaxPotentialBlockSize	
K	block column K		
I	block row I		
$mybrID$	block row id		
$ready\_x$	receive buffer for x	nvshmem_malloc, size is $mz \cdot N_g$	
$ready\_lsum$	receive buffer for block rows	nvshmem_malloc, size is $2 \cdot mz \cdot N_r$	
$flag\_x$	notification buffer for x	nvshmem_malloc, size is N_g	
$flag\_lsum$	notification buffer for $lsum$	nvshmem_malloc, size is $2 \cdot N_r$	
M_c	mask vector for block column broadcast	cuda_malloc, size is N_g	
M_r	mask vector for block row reduction	cuda_malloc, size is $2 \cdot N_r$	
	NVSHMEM Communication APIs	Descriptions	Callsite Scope
Sender	nvshmem_double_put_nbi_block	send x subvector	thread blocks
	nvshmem_double_put_nbi()	send $lsum$	threads
	nvshmem_fence()	ensure the orders	
	nvshmemx_int_signal()	send notification	threads
Receiver	nvshmem_int_wait_until_any()	receive notification	threads

The `WAIT` kernel in *stream[0]* (lines 26-53) uses *nvshmemx_wait_until_any* to receive the data completion notification from the senders. We launch two thread blocks with $maxTH$ threads in each thread block. Here $maxTH$ is the maximum block size returned by *cudaOccupancyMaxPotentialBlockSize*. The first thread block is used for column broadcasts while the second is used for row reductions. Consider the thread block used for block column broadcasts (line 28-33), recall that we have calculated a mask vector M_c in the pre-processing phase. Here we distribute this mask among the threads in this thread block, and let each thread wait for a sub-

set of all the block columns. Figure 2 describes the mask distribution among threads. Assuming we have a mask of length 17 (i.e., $N_g = 17$), and one thread block is launched for column broadcast, with five threads in it. $mask[i] = 0$ means the GPU expects one message in block column i, and 1 means that no message is needed. There are 10 messages in total, and we let each thread wait for two messages to balance the number of messages to be waited across the threads. $flag\_x$ is a bit vector including a bit for each column, and it is a NVSHMEM buffer which will be updated by other GPUs. While the mask M_c is a local buffer to manage the scope of waiting columns for each thread.

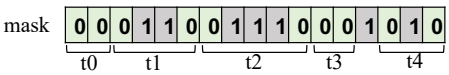


Figure 2: Each thread (ti) in `WAIT` kernel has its own entry in mask.$mask[i] = 0$ means block column i waits for one message. 1 means no message is needed.

In the row reduction (line 34-52), each block row has two bits in M_r and $flag\_lsum$ since one block row receives two messages at most. We initialize the two bits to zeros if one block row expects messages, and then distribute M_r across threads. Once received all required message, the corresponding thread will accumulate the local summation, and then send to its parent according to the binary reduction tree if the counter $fmod(I)$ becomes zero. Recall that each block row receives two messages at most, and we initialize two bits to zeros no matter it expects one message or two messages. Now the question is how receivers know whether they have received all required messages. A counter $cnt[I]$ (line 38) for each row I is computed when building communication trees on CPUs. Thus, block row I receives all required messages when the summation of the two bits in buffer $flag\_lsum$ equals to $cnt[I]$.

In contrast to traditional bulk synchronous parallel GPU implementations, our multi-GPU SpTRSV design can be considered as a message-driven algorithm. If a received message is a subvector of x, the GPU forwards the message according to the binary broadcast tree (line 16) before performing local accumulation. Otherwise if the message is $lsum$, the GPU accumulates it to the local sum. Once the counter $fmod$ becomes zero, the GPU has received all required messages and executed all its assigned GEMVs, then that GPU forwards the $lsum$ according to the binary reduction tree.

Inter-Stream Communication: The two kernels in two streams need to interact with each other so that (1) the `SOLVE` kernel can execute GEMV when the expected x subvector is received, and (2) both kernels need to track the counter $fmod$ in order to send the $lsum$ in time.

Recalled that the `WAIT` kernel uses $flag\_x$ (column broadcast) to receive the notification from the senders. That buffer is located in GPU global memory. Therefore, the `SOLVE` kernel can access $flag\_x$ simultaneously in another stream as Figure 3 shows (corresponding to Algorithm 1 line 15). Thread blocks that need to receive messages in the `SOLVE` kernel keep reading the $flag\_x$ buffer until the corresponding location in that buffer is updated by the `WAIT` kernel.

The counter $fmod$ is used to maintain the data dependency in a row reduction and is also located in GPU global memory so that the two kernels can access $fmod$ concurrently. The `SOLVE` kernel atomically decrements $fmod$ once it finishes the local inner-products. Meanwhile, the `WAIT` kernel also atomically decrements $fmod$ once it receives non-local messages as visualized in Figure 3 (corresponding to Algorithm 1 lines 20 and 48). When $fmod$ becomes zero, the corresponding thread, either the `SOLVE` kernel or the `WAIT` kernel, will send its local summation to its parent according to the binary reduction tree.

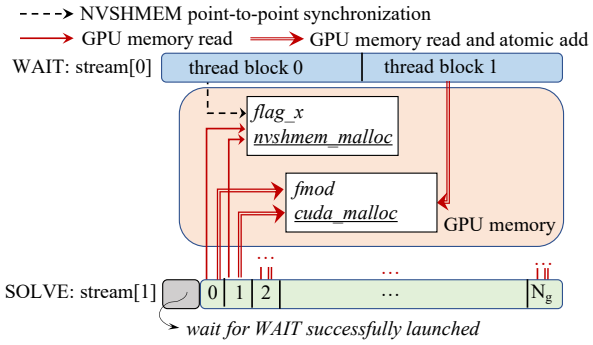


Figure 3: Communication between two streams. The two kernels use $flag\_x$ and $fmod$ to maintain data dependencies.

Extensibility: Our work can be migrated to other GPU-accelerated architectures that support GPU-initiated one-sided communications, e.g., AMD GPUs with ROC_SHMEM [20, 21] which has a similar syntax to NVSHMEM.

5 SpTRSV Performance Model for GPUs

We extend the critical path model [12] to GPUs to explain the observed performance, and explore the key features of matrices to estimate the potential performance benefits when using multiple GPUs.

The model is based on the critical path analysis which follows the task dependency graph of the sparse matrix using the well-known level-set method [13, 14] with a breadth-first search [15]. We further extend this critical path model to GPUs by (1) refining the DAG nodes in each level according to messaging patterns, and

Algorithm 2 SpTRSV performance model for GPUs

Application Inputs:
$Nsuper$: number of supernodes
w_i: width of block i (bytes)
h_i: height of block i (bytes)
Architecture Inputs:
bw_p: peak memory bandwidth per GPU
bw_g: GPU-GPU data transfer bandwidth (bytes/GEMV/second)
L_g: GPU-GPU data transfer latency
P: number of GPUs (processes)
MAX_TB: Number of thread blocks per GPU that can achieve the peak memory bandwidth
Outputs:
T_{comp_p}: TRSV/GEMV time of GPU p
T_{comm_p}: GPU-GPU data transfer time of GPU p
T_{tot}: total SpTRSV time

1: **procedure** MODELING
2: Analyze Critical path: find DAG levels and DAG nodes
3: Count $out_{b\_l\_p}$ (broadcast) of each DAG level l for GPU p
4: Count $out_{r\_l\_p}$ (reduction) of each DAG level l for GPU p
5: $bw_m = \frac{bw_p}{Nsuper/P}$ ▷ memory bandwidth per thread block (bytes/second)
6: **If** (bw_m < 1.3 GB/s) bw_m=1.3 GB/s;
7: **If** (bw_m > 5.2 GB/s) bw_m=5.2 GB/s;
8: **for** each DAG level l **do**
9: $mynodes_{l\_p} = 0$, $mybytes_{l\_p} = 0$
▷ $mynodes_{l\_p}$: number of active DAG nodes in level l of GPU p
10: **for** each DAG nodes i **do**
11: $mynodes_{l\_p} += 1$
12: $mybytes_{l\_p} += w_i \cdot h_i$
13: **if** $mynodes_{l\_p}$ >= MAX_TB **then** ▷ reach memory capacity
14: $T_{comp_{l\_p}} += \frac{mybytes_{l\_p}}{bw_p}$
▷ $T_{comp_{l\_p}}$: compute time of level l in GPU p
15: $mybytes_{l\_p} += 0$
16: $mynodes_{l\_p} += 0$
17: **end if**
18: **if** $mynodes_{l\_p}$ < MAX_TB && ends level l **then**
19: **if** $tune == 1$ **then**
20: $bw_m = \frac{bw_p}{mynodes_{l\_p}}$
21: **If** (bw_m < 1.3 GB/s) bw_m=1.3 GB/s;
22: **If** (bw_m > 5.2 GB/s) bw_m=5.2 GB/s;
23: **end if**
24: $T_{comp_{l\_p}} += \frac{mybytes_{l\_p}}{bw_m * mynodes_{l\_p}}$
25: **end if**
26: **end for**
27: **end for**
28: **for** each DAG level l **do**
29: **for** each supernode column c in DAG level l **do**
30: **if** $out_{b\_l\_p\_c}! = 0$ **then** ▷ broadcast
31: $out_{nb\_l\_p\_c} = out_{b\_l\_p\_c} > 2? \log_2(out_{b\_l\_p\_c}) : out_{b\_l\_p\_c}$
▷ $out_{nb\_l\_p\_c}$: non-overlapped messages in column c on level l in GPU p
32: $T_{comm_{l\_p}} += L_g + out_{nb\_l\_p\_c} \cdot \frac{w_i}{bw_g}$
33: **end if**
34: **end for**
35: **for** each supernode row c in DAG level l **do**
36: **if** $out_{r\_l\_p\_c}! = 0$ **then** ▷ broadcast
37: $out_{nr\_l\_p\_c} = out_{r\_l\_p\_c} > 2? \log_2(out_{r\_l\_p\_c}) : out_{r\_l\_p\_c}$
▷ $out_{nr\_l\_p\_c}$: non-overlapped messages of row c on level l in GPU p
38: $T_{comm_{l\_p}} += L_g + out_{nr\_l\_p\_c} \cdot \frac{h_i}{bw_g}$
39: **end if**
40: **end for**
41: **end for**
42: Find GPU p_{max} ($\sum_{l=0}^{L} T_{comm_{l\_p}} + T_{comp_{l\_p}}$) who has the longest time
43: $T_{tot} = T_{p_{max}}$
44: **If** ($\frac{T_{single\ GPU}}{T_{tot}} > P$) re-model with $tune = 1$
45: **end procedure**

(2) taking memory scaling bandwidth into consideration when modeling GEMV and TRSV time.

The computation dependency of SpTRSV can be precisely expressed by a DAG. Let us consider a L matrix which is factorized via `SuperLU_DIST` with METIS ordering for fill-in reduction [22]. Thus, DAG nodes refer to dense matrix-vectors, and edges between DAG nodes represent data dependencies. DAG nodes in the same level can be solved concurrently, and DAG levels must be solved sequentially. When it comes to multi-GPU SpTRSV, we can further remove the edges between DAG nodes that assigned to the same GPU. This is because thread blocks can each be executed independently and thus may execute in parallel. That is to say, DAG nodes located in one GPU can be solved concurrently by multiple thread blocks. Ultimately, the edges in the refined DAG represent only GPU-GPU messages.

Algorithm 2 details the extended GPU SpTRSV model. The SpTRSV time is modeled based on the refined DAG. The matrix features required to build the model are number of supernodes (N_{super}), number of nonzeros of each DAG node (w_i and h_i). The computation time of each GPU (process) is the accumulation time of DAG levels. In each level, memory bandwidth scales with the number of DAG nodes until the aggregate memory bandwidth reaches the peak (line 10-14). The empirical HBM bandwidth (bw_p) is 828 GB/s [23]. According to the white paper of NVIDIA Tesla V100 accelerator (V100 [2]), the maximum number of thread blocks per V100 is $\frac{80SMs \cdot 64warps}{8warps} = 640$, where 8 warps per thread block is based on our design. Therefore, we set the lower bound of memory bandwidth per thread block (bw_m) to 1.3 GB/s. However, the number of active thread blocks can be much smaller than the maximum of 640 due to either data dependencies or hardware limit. That is to say, in some cases, the bw_m can be larger than 1.3 GB/s. Let's consider the question of how many thread blocks can leverage a full bandwidth of a SM with dependencies. Ideally, the smallest number is two. One thread block spin waits the dependency and the other one can perform other independent computation work. Thus, the bandwidth per thread block $bw_m = \frac{828GB/s}{80SMs \cdot 2warps} = 5.2$ GB/s. Correspondingly, MAX_TB=$80 \cdot 2 = 160$. Ultimately, the upper and lower bound of bw_m is 5.2 GB/s and 1.3 GB/s.

The communication time is modeled according to the number of messages and message size of each DAG node on the critical path. We count the $out_{b\_l\_p\_c}$ (the number of broadcast messages happened in column c of level l in GPU p) and $out_{r\_l\_p\_c}$ (the number of reduction messages occurring in row c on level l in GPU p) according to the process decomposition. In column broadcast, each message has a size of the width

of block column i (w_i). The latency of multiple sends can be overlapped because all the messages are coming from the same producer. Let us assume there are P processes participating in the block column broadcast. When using a binary communication tree, each process that participates in the block column broadcast sends at most two messages to its children. This reduces the send message count of the corresponding process by $log_2 P$. Ultimately, for each GPU the accumulated communication time of each DAG level is the final communication time. The communication time on each level is estimated using the number of non-overlapped messages in GPU p (line 30-33). The row reduction follows the same manner. Each message size equals the height of block rows i (h_i). Thus, each GPU has a total SpTRSV time which equals to the accumulated time of computation and communication of DAG nodes on its critical path. We then take the longest SpTRSV time among the GPUs as the final SpTRSV time, and the critical path of that GPU is the final critical path.

We introduce a refinement feature in the model (line 44). Once we model the total SpTRSV time of using P GPUs, we compare the T_{tot} with the single GPU time. If the speedup is larger than the superlinear speedup P, we believe such discrepancy is due to the optimized memory bandwidth per thread block. Recall that by default every 160 DAG nodes (thread blocks) can achieve the peak memory bandwidth. Thus, at the end of DAG levels, each thread block may achieve a higher memory bandwidth than the initialized $bw_m = \frac{bw_p}{Nsuper/P}$ if the number of unsolved DAG nodes is less than the number of supernodes per GPU. (lines 19-23). In the refinement phase, we turn off that memory bandwidth adjustment, and use the the initialized bw_m instead of that optimized one.

BW_g and L_g are parameterized by benchmarked message sizes using a round-trip ping pong benchmark. When estimating the communication time, we round up (optimistic) the message size to the next power of two to match the corresponding BW_g in the model.

6 Results

In this section, we report experiment results and analysis of our multi-GPU SpTRSV using CUDA streams, including strong scaling performance evaluation with different process decompositions, and the analysis of the observed performance. We then discuss the key matrix features to determine the potential benefit of a matrix to use multiple GPUs.

6.1 Experimental Setup: Results presented in this paper were obtained on the GPU-accelerated partition on Summit at OLCF. Each of the Summit nodes contains two IBM POWER9 processors and six NVIDIA Tesla V100 accelerators. The GPUs within a node are connected by NVIDIA's NVLink interconnect. Summit nodes are connected using EDR InfiniBand interconnect. In all experiments, the SpTRSV runs on GPUs using double-precision real matrices. We use CUDA 10 and NVSHMEM 1.1.3 with GDRcopy 2.0.

Table 2 presents the key features of the matrices used in this paper. These matrices have also been used in various computational research [24–27]. Matrix S1 comes from M3D-C1, a fusion simulation code used for magnetohydrodynamics modeling of plasma [25]. All other matrices are publicly available through the SuiteSparse Matrix Collection [28]. The selected matrices cover a wide range of matrix properties (i.e., matrix size, sparsity structure, the number of level-sets, and application domain). The matrices are first factorized via `SuperLU_DIST` with METIS ordering for fill-in reduction [22]. The resultant lower triangular matrices are used with the proposed multi-GPU implementation.

6.2 Scalability Evaluation: Figure 4 shows our speedups (using the optimal process decomposition in each concurrency) compared to the single GPU version of cusparse_csrsv2(). Our single GPU implementation outperforms cusparse_csrsv2() by up to 1.9× speedup. Our multi-GPU SpTRSV provides a performance improvement of up to 6.1× when using twelve GPUs. Therefore, our implementation enables GPU-accelerated, distributed memory computing via NVSHMEM.

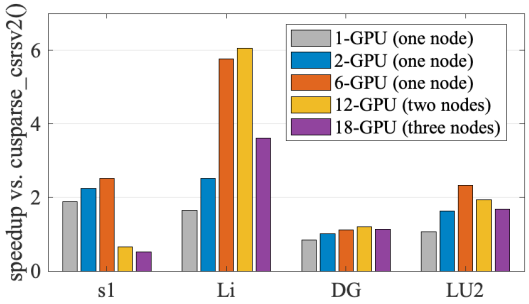
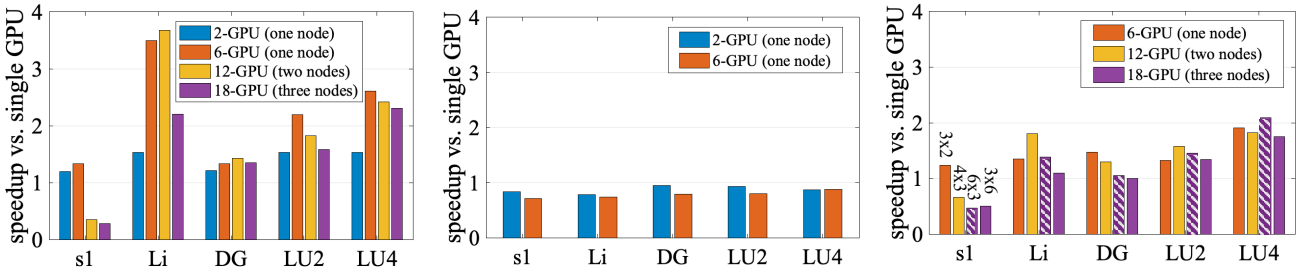


Figure 4: Multi-GPU Lower triangular solve time compared to *cusparse_csrsv2()*. Our implementation achieves up to 6.1× speedup with a scale from single GPU to eighteen GPUs (three Summit nodes).

Figure 5a shows the lower triangular solve speedup using a P×1 process decomposition (column broadcast) compared to our single GPU implementation. Our multi-GPU implementation using CUDA streams attains a speedup of up to 3.7× when using up to twelve GPUs (two nodes). Figure 5b presents the lower triangular solve speedup using a 1×P process decomposition (row reduction) on one Summit node. In this

Table 2: Test matrices.

Matrix	#supernodes	nnz in L	Levels	Maximum Speedup			
				2 GPUs	6 GPUs	12 GPUs	18 GPUs
S1	9,827	8.80E+08	388	1.2×	1.3×	0.7×	0.8×
DG_GrapheneDisorder (DG)	2,000	9.66E+08	199	1.2×	1.3×	1.4×	1.3×
LU_C_BN_C_2by2 (LU2)	1,216	8.54E+08	264	1.5×	2.2×	1.8×	1.9×
LU_C_BN_C_4by2 (LU4)	2,383	1.87E+09	320	1.5×	2.6×	2.4×	2.2×
Li4244 (Li)	362	5.18E+08	188	1.5×	3.5×	3.7×	2.7×



(a) P×1 process decomposition (column broadcast)

(b) 1×P process decomposition (row reduction).

(c) 2D process decomposition

Figure 5: Multi-GPU lower triangular solve time compared to our single GPU implementation. Our Multi-GPU implementation achieves up to 3.5× speedup on one Summit node, and achieves up to 3.7× speedup between two to three nodes using a P×1 process decomposition.

case, the performance of the single GPU outperforms the multi-GPU implementation. Figure 5c shows the speedups of using a 2D process decomposition (both column broadcast and row reduction). Performance results demonstrate that our implementation favors row parallelism over column parallelism, because the latter involves row reductions which are more expensive than column broadcasts.

Our multi-GPU SpTRSV is often able to exploit multiple GPUs on one node. On the other hand, performance is challenged when using GPUs that span multiple nodes but rarely falls off a cliff. Thus, when limited GPU memory capacity necessitates spreading a matrix over multiple nodes, our implementation can deliver acceptable performance whereas a single GPU implementation would be incapable of holding the matrix.

It is worth mentioning that although SpTRSV is challenged beyond twelve GPUs, it can scale to 4,096 processes on CPUs [12]. This is an artifact of faster GPU computational performance being offset by a much lower inter-node messaging performance. Specifically, a V100 provides 7 TFLOP/s of performance while a KNL core only provides about 0.4 TFLOP/s. At the same time, the one-sided NVSHMEM messaging performance is about 7× slower than foMPI which is used in work [12]. According to the microbenchmarks of our largest typical message size 1024 bytes, work [12] uses a network with 545 MB/s bandwidth, while the current NVSHMEM bandwidth is only 75 MB/s.

6.3 Strong Scaling Performance Analysis: We first discuss two key observations in Section 6.2: (1) speedups on multiple nodes are diminished, and (2) the selected matrices demonstrate very different scaling behaviours. We then demonstrate (3) the predictive ability of our model, which can help users to determine the number of GPUs that produces the fastest run time.

Inter-GPU Networking Performance. The smaller speedups on multiple nodes are due to the low performance of the inter-node GPU-GPU network. Figure 6 highlights the NSVHMEM SEND bandwidth between two GPUs (processes) of intra-socket, intra-node and inter-node using three different callsite scopes: **Thread block:** Use all threads in thread blocks to put data to the target GPU (process) by `nvshmem_double_put_nbi_block`, and then perform a `nvshmem_fence`. Finally, use thread 0 to send notification via `nvshmemx_int_signal`. **Warp:** Use one warp in each thread block to put data to the target GPU with `nvshmemx_double_put_warp`, and the rest remain the same with *thread block*. **Thread:** Use one thread in each thread block to put data to the target GPU with `nvshmem_double_put`, and the rest remain the same with *thread block*.

The GPU-GPU bandwidth using *thread blocks* outperforms the performance using warps and threads by 2× and 9× on average. Using thread blocks or warps deliver the same performance as using threads for inter-node communication. This is because only one single thread in a thread block/warp can issue an RMA write operation to the destination GPU over InfiniBand.

Ultimately, one should remember that the performance of one-sided messaging libraries can vary sub-

stantially. For example, on the Cray Aries network, Cray's one-sided implementation is 2.7× slower than Cray's two-sided [12] yet ETH's foMPI is 3× faster. Here, one-sided NVSHMEM is 2.3× slower than IBM Spectrum MPI over InfiniBand network on Summit [29].

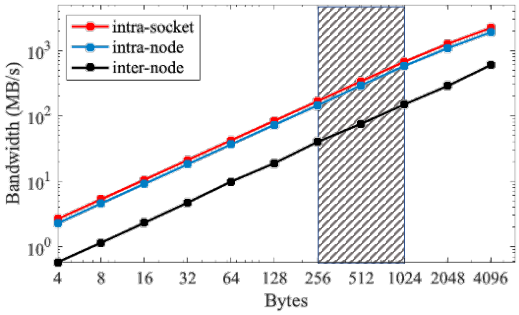


Figure 6: NVSHMEM SEND (thread block) bandwidth using two GPUs on Summit. The shadowed stripe highlights the typical message size in SpTRSV of 256 bytes to 1,024 bytes. Intra-socket NVSHMEM SEND outperforms intra-node NVSHMEM SEND (avg. 1.2×) and inter-node NVSHMEM SEND (avg. 3.9×).

Process Decomposition and Performance. Recall that the 1×P implementation leverages only a single thread to perform NVSHMEM SENDs (Algorithm 1 line 21 and 49), the overall SpTRSV performance using a 1×P implementation (Figure 5b) is limited by the resultant low NVSHMEM SEND throughput. In a 2D process decomposition, as we increase the number of GPUs participating in the row reduction, the number of messages (single thread NVSHMEM SEND) in the row reduction increases, and thus the performance decreases. Essentially, the smaller speedup for the 2D process decomposition compared to the P×1 implementation is again due to the low NVSHMEM SEND throughput (single thread) in row reductions.

Matrix Properties and Performance. Matrix DG and Li have a similar number of DAG levels: 199 and 188, respectively. Since DG has more nonzeros (966 million nonzeros) than Li (518 million nonzeros), one might assume that DG scales better than Li. However, the reality is that Li achieves 2.7× speedup on average (up to 3.7× on twelve GPUs) while DG has 1.3× speedup on average (up to 1.4× on six GPUs). Such a discrepancy is due to ignoring communication and memory bandwidth.

According to the model in Algorithm 2, matrix Li has 162 message on the critical path using two GPUs, 270 messages using six GPUs, and 292 message using twelve GPUs, while DG has 1,000, 898 and 571 messages, respectively. Hence, one can immediately understand that the large number of messages of DG makes its scaling performance worse than matrix Li.

Another aspect is the achieved memory bandwidth per thread block. Matrix Li has only 362 supernodes in total. Therefore, it can achieve 4.6 GB/s ($\frac{828}{362/2}$) when using two GPUs and 5.2 GB/s (upper bound of memory bandwidth per thread block, Algorithm 2 line 7) when using more than two GPUs. While DG achieves only 1.3 GB/s (lower bound of memory bandwidth per thread block, Algorithm 2 line 6) with 1,000 supernodes per GPU when using two GPUs, and 4.0 GB/s when using twelve GPUs (highest speedup, 1.4×). A smaller number of supernodes (thread blocks) per GPU produces less memory contention. Thus, each thread block can achieve a higher memory bandwidth.

Ultimately, matrix Li achieves the best scaling performance among the selected matrices. This is due the weak data dependency (only 188 DAG levels and a small number of messages on the critical path) and high memory bandwidth per thread block (5.2 GB/s). Even though matrix DG has a similar number of levels as matrix Li, it does not scale as well as Li due to the limited memory bandwidth per thread block and relatively larger number of messages on the critical path.

Model prediction. Figure 7 visualized the modeled time and measured times of the S1 matrix (achieves the smallest speedup at scale, up to 1.3×, among the matrices in Table 2) and the Li matrix (highest speedup, up to 3.7×). The total modeled SpTRSV time equals the accumulation of the communication time on the critical path (yellow bar) and the computation time on the critical path (blue bar). In addition to understanding and explaining observed performance, the SpTRSV model can also help identify the number of GPUs that produces the fastest run time.

The overall sweet spot of S1 is six GPUs within a node. From the model, we see that the numbers of messages on the critical paths are very similar: 7,922 messages (six GPUs) and 7,408 message (twelve GPUs), but the network bandwidth decreases as more nodes are used. Ultimately, the resultant low inter-node NVSHMEM SEND throughput makes the run time of S1 increase when using more than one node.

Unlike the S1 matrix, the communication time of Li only takes 33% when using up to eighteen GPUs. It's the computation time dominants the total run time. From one to six GPUs, we see a nearly linear reduction in computation time of matrix Li because the achieved memory bandwidth increases: 1.3 GB/s per thread block using one and two GPUs (362 supernodes using one GPU, and 181 supernodes per GPU using two GPUs) and 5.2 GB/s per thread block using six GPUs (55 supernodes per GPU).

Rather than using average parallelism of a matrix as metrics, our model incorporates the number of levels

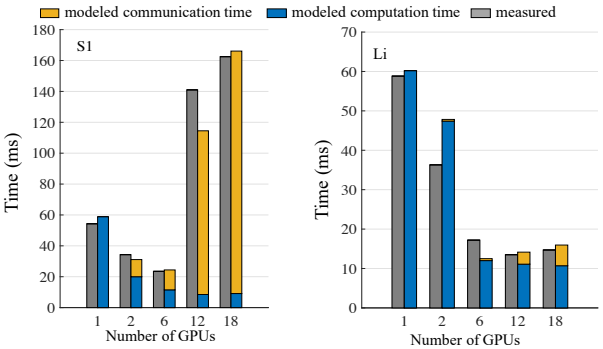


Figure 7: The performance model can help identify the number of GPUs that produces the fastest run time. The total modeled SpTRSV time equals the accumulation of the blue bar and the yellow bar.

(dependency), number of non-zeros on the critical path, and architecture features (memory bandwidth, and network bandwidth). The model highlights that the improvement in memory bandwidth per thread block (not just aggregate bandwidth) can help reduce computation time for matrices like S1, while improving NVSHMEM SEND throughput can help attain better scalability for matrices like Li.

7 Related Work

Exploring high performance SpTRSV is becoming ever more crucial on GPU-accelerated architectures. Most existing parallel GPU triangular solvers focus on optimizing single GPU performance [6, 30–33]. Due to the complex data dependencies in SpTRSV, algorithm optimization has been mainly based on the level-set methods and color-set methods for various parallel architectures. Additionally, there is research focused on optimizing the block structure [34], or analyzing nonzero layout and selecting the best sparse kernels by using machine learning and deep learning methods [35–37]. Our work explore the benefits of using multiple GPUs, and we believe that our proposed method of coupled producer-consumer parallelism using CUDA streams can bring insightful experience for DAG-based computations on emerging accelerated architectures.

The existing work of distributed-memory SpTRSV have mainly conducted on CPU platforms. Using 1D or 2D process layouts to improve load balance is discussed in work [38–40]. An asynchronous binary tree is proposed to reduce the communication latency [17]. Venkat et al. [41, 42] developed several techniques that generate wavefront parallelization with faster level-set scheduling. One-sided communication is used in work [12] to implement a synchronization-free task queue to manage messages between producer-consumer pairs. Xie et al. propose a multi-GPU SpTRSV using a 1×P process decomposition [43]. The inter-GPU communications are performed via NVSHMEM warp-level get, and it outperforms *cusparse_csrsc2()* by up to 3.2× on average using 16 GPUs on one DGX-2 node with 20% parallel efficiency. In comparison, our new algorithm achieves up to 2.9× speedup on average using six GPUs on one Summit node with 58% parallel efficiency. Hamidouche et al. implement a multi-GPU SpTRSV on AMD GPUs using ROC_SHMEM [21]. They achieved up to a 3.7× speedup compared to a baseline that used intra-kernel communication (rely on CPU threads to perform network operations on behalf of the GPU) rather than the optimal single GPU solution. It is worth mentioning that our work starts from a faster baseline. In addition, comparisons of AMD ROC_SHMEM on AMD GPUs using a column-based approach against NVIDIA NVSHMEM on NVIDIA GPUs using a supernodal approach imperil meaningful insights as too many variables were changed.

8 Conclusion

We use CUDA streams to perform coupled producer-consumer parallelism in multi-GPU SpTRSV. Over the range of two to eighteen GPUs, our multi-GPU SpTRSV implementation improve solve time by up to 3.7× compared to our single GPU implementation, and up to 6.1× compared to *cusparse_csrsv2()*. In order to assess our observed performance relative to machine capabilities and matrix features, we constructed a critical path performance model. Our SpTRSV model endows users with far greater insights as to how different aspects of multi-GPU architectures and matrix features constrain performance, e.g., the limited achieved memory bandwidth per thread block constrains the scaling performance of matrix S1, while low NVSHMEM SEND throughput constrains the performance when using 1×P and 2D decompositions.

Fusion simulations like M3DC1 [25] and NIMROD [44], solve highly ill-conditioned linear systems. One successful method is to use GMRES with a block-Jacobi preconditioner, where each diagonal block is solved by SuperLU_DIST. There is no inter-block communication within the preconditioner. Our results highlight the computational importance of keeping block size small enough so that it fits on a single node.

In the future, we will continue to refine the model to highlight the performance nuances, use the model to identify potentially superior process mappings, and port our supernodal-based triangular solver to other emerging accelerators. More broadly, we will explore the value of our multi-CUDA stream approach in other domains.

References

[1] Top500 Highlights - November 2020. URL https://www.top500.org/lists/top500/2020/11/highs/.

[2] Nvidia Tesla. V100 gpu architecture. *Online verfügbar unter http://images. nvidia. com/content/volta-architecture/pdf/volta-architecture-whitepaper. pdf, zuletzt geprüft am*, 21, 2018.

[3] István Z Reguly, Gihan R Mudalige, Carlo Bertolli, Michael B Giles, Adam Betts, Paul HJ Kelly, and David Radford. Acceleration of a full-scale industrial CFD application with OP2. *IEEE Transactions on Parallel and Distributed Systems*, 27(5): 1265–1278, 2016.

[4] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on uncertainty in artificial intelligence (UAI)*, pages 340–349, 2010.

[5] Maxim Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu. *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011*, 1, 2011.

[6] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S Duff, and Brian Vinter. A synchronization-free algorithm for parallel sparse triangular solves. In *European Conference on Parallel Processing*, pages 617–630. Springer, 2016.

[7] NVIDIA NVSHMEM Documentation. URL https://docs.nvidia.com/hpc-sdk/nvshmem/index.html.

[8] Jeff R Hammond, Sayan Ghosh, and Barbara M Chapman. Implementing OpenSHMEM using MPI-3 one-sided communication. In *Workshop on OpenSHMEM and Related Technologies*, pages 44–58. Springer, 2014.

[9] M. Sourouri, T. Gillberg, S. B. Baden, and X. Cai. Effective multi-gpu communication using multiple cuda streams and threads. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 981–986, 2014. doi: 10.1109/PADSW.2014.7097919.

[10] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009.

[11] Nan Ding and Samuel Williams. An instruction roofline model for gpus. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 7–18. IEEE, 2019.

[12] Nan Ding, Samuel Williams, Yang Liu, and Xiaoye S Li. Leveraging one-sided communication for sparse triangular solvers. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*, pages 93–105. SIAM, 2020.

[13] Edward Anderson and Youcef Saad. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing*, 1 (01):73–95, 1989.

[14] Joel H Saltz. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM journal on scientific and statistical computing*, 11(1):123–144, 1990.

[15] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[16] James W Demmel, Stanley C Eisenstat, John R Gilbert, Xiaoye S Li, and Joseph WH Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[17] Yang Liu, Mathias Jacquelin, Pieter Ghysels, and Xiaoye S Li. Highly scalable distributed-memory sparse triangular solution algorithms. In *Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*, pages 87–96. SIAM, 2018.

[18] Alexandros V Gerbessiotis and Leslie G Valiant. Direct bulk-synchronous parallel algorithms. *Journal of parallel and distributed computing*, 22(2): 251–267, 1994.

[19] An Introduction to CUDA-Aware MPI. URL https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/.

[20] ROC_SHMEM. https://github.com/ROCm-Developer-Tools/ROC_SHMEM, 2020.

[21] Khaled Hamidouche and Michael LeBeane. Gpu initiated openshmem: correct and efficient intra-kernel networking for dgpus. In *Proceedings of*

the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 336–347, 2020.

[22] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998. doi: 10.1137/S1064827595287997. URL https://doi.org/10.1137/S1064827595287997.

[23] Charlene Yang, Thorsten Kurth, and Samuel Williams. Hierarchical roofline analysis for gpus: Accelerating performance optimization for the nersc-9 perlmutter system. *Concurrency and Computation: Practice and Experience*, 32(20):e5547, 2020.

[24] Ulrike Baur, Peter Benner, Andreas Greiner, Jan G Korvink, Jan Lienemann, and Christian Moosmann. Parameter preserving model order reduction for MEMS applications. *Mathematical and Computer Modelling of Dynamical Systems*, 17(4):297–317, 2011.

[25] S C Jardin, N Ferraro, X Luo, J Chen, J Breslau, K E Jansen, and M S Shephard. The M3D-C1 approach to simulating 3D 2-fluid magnetohydrodynamics in magnetic fusion experiments. *J. Phys. Conf. Ser.*, 125(1):012044, 2008. URL http://stacks.iop.org/1742-6596/125/i=1/a=012044.

[26] Alexander Ludwig. The Gauss–Seidel–quasi-Newton method: A hybrid algorithm for solving dynamic economic models. *Journal of Economic Dynamics and Control*, 31(5):1610–1632, 2007.

[27] Gary Kumfert and Alex Pothen. Two improved algorithms for envelope and wavefront reduction. *BIT Numerical Mathematics*, 37(3):559–590, 1997.

[28] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011. ISSN 0098-3500. doi: 10.1145/2049662.2049663. URL http://doi.acm.org/10.1145/2049662.2049663.

[29] Summit User Guide. URL https://docs.olcf.ornl.gov/systems/summit_user_guide.html.

[30] Jiya Su, Feng Zhang, Weifeng Liu, Bingsheng He, Ruofan Wu, Xiaoyong Du, and Rujia Wang. Capellinisptrsv: A thread-level synchronization-free sparse triangular solve on gpus. In *49th International Conference on Parallel Processing-ICPP*, pages 1–11, 2020.

[31] Ruipeng Li and Chaoyu Zhang. Efficient parallel implementations of sparse triangular solves for gpu architectures. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*, pages 106–117. SIAM, 2020.

[32] Zhengyang Lu, Yuyao Niu, and Weifeng Liu. Efficient block algorithms for parallel sparse triangular solve. In *49th International Conference on Parallel Processing-ICPP*, pages 1–11, 2020.

[33] Ernesto Dufrechou and Pablo Ezzatti. A new gpu algorithm to compute a level set-based analysis for the parallel solution of sparse triangular systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 920–929. IEEE, 2018.

[34] Ehsan Totoni, Michael T Heath, and Laxmikant V Kale. Structure-adaptive parallel solution of sparse triangular linear systems. *Parallel Computing*, 40 (9):454–470, 2014.

[35] Guangming Tan, Junhong Liu, and Jiajia Li. Design and implementation of adaptive spmv library for multicore and many-core architecture. *ACM Transactions on Mathematical Software (TOMS)*, 44(4):46, 2018.

[36] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. Bridging the gap between deep learning and sparse matrix format selection. In *ACM SIGPLAN Notices*, volume 53, pages 94–108. ACM, 2018.

[37] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *ACM SIGPLAN Notices*, volume 48, pages 117–126. ACM, 2013.

[38] Ehsan Totoni, Michael T. Heath, and Laxmikant V. Kale. Structure-adaptive parallel solution of sparse triangular linear systems. *Parallel Computing*, 40(9):454 – 470, 2014. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2014.06.006. URL http://www.sciencedirect.com/science/article/pii/S0167819114000799.

[39] Xiaoye S. Li and James W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–17, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-89791-984-X. URL http://dl.acm.org/citation.cfm?id=509058.509092.

[40] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Softw.*, 29(2):110–140, June 2003. ISSN 0098-3500. doi: 10.1145/779359.779361. URL `http://doi.acm.org/10.1145/779359.779361`.

[41] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. Automating wavefront parallelization for sparse matrix computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 41. IEEE Press, 2016.

[42] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *ACM SIGPLAN Notices*, volume 50, pages 521–532. ACM, 2015.

[43] Chenhao Xie, Jieyang Chen, Jesun S Firoz, Jiajia Li, Shuaiwen Leon Song, Kevin Barker, Mark Raugas, and Ang Li. Fast and scalable sparse triangular solver for multi-gpu based hpc architectures. *arXiv preprint arXiv:2012.06959*, 2020.

[44] C.R. Sovinec, A.H. Glasser, T.A. Gianakon, D.C. Barnes, R.A. Nebel, S.E. Kruger, S.J. Plimpton, A. Tarditi, M.S. Chu, and the NIMROD Team. Nonlinear magnetohydrodynamics with high-order finite elements. *J. Comp. Phys.*, 195:355, 2004.