NVIDIA

ACCELERATING SPARSE DIRECT SOLVERS: STRATEGIES FOR HIGH PERFORMANCE ON NVIDIA GPUS

Samuel Rodriguez, Anton Anders, Kirill Voronin, Alexander Kalinkin

CUDA DIRECT SPARSE SOLVER

What is it?

- Goal: unlock the potential of NVIDIA GPU HW for solving (large) sparse linear systems with direct methods and prove wrong the conventional wisdom “GPU is not good for direct linear solvers” by utilizing both high memory bandwidth and compute power of GPU
    - Eternal question: direct or iterative methods? Answer: depends on the app
- Current state: standalone CUDA Math library supporting
    - fp32/fp64 real/complex matrices + int32 indexing, all matrix types, different reordering schemes, pivoting controls, output stats
    - Linux + x86, Linux + SBSA (Grace), Windows + x86
- Customers from application domains: circuit/aerospace/CFD simulations, SLAM, robotics, autonomous driving, CAE and more

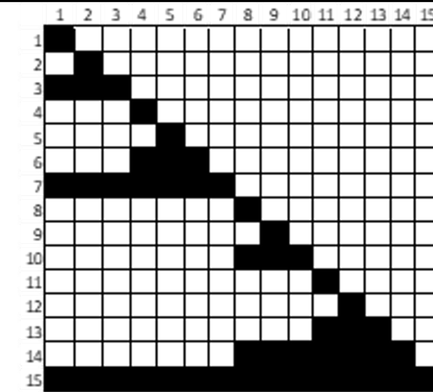
NVIDIA CONFIDENTIAL. DO NOT DISTRIBUTE.

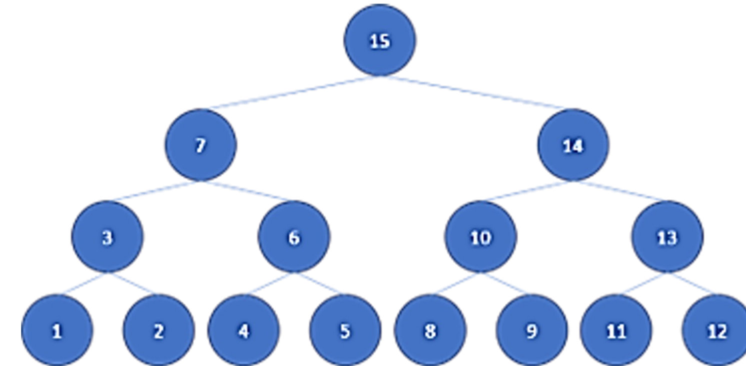
CUDA DIRECT SPARSE SOLVER

Algorithm Overview

Phase 1: Reordering

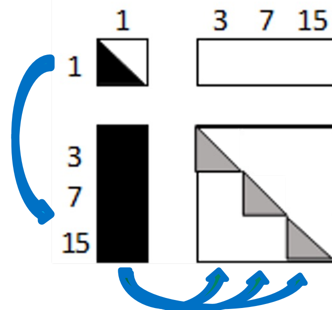
Representing initial matrix as a binary graph to extract more parallelism and reduce memory requirements.





Phase 2: Factorization

Factorizing reordered matrix as A = LU (lower triangular x upper triangular matrices)



Simple factorization scheme that consists of 3 main workloads:

1) factorization of the diagonal block
2) TRSM for sub-diagonal blocks
3) GEMM to update next blocks

Phase 3: Solving

Solving the equivalent system with lower and upper triangular factors

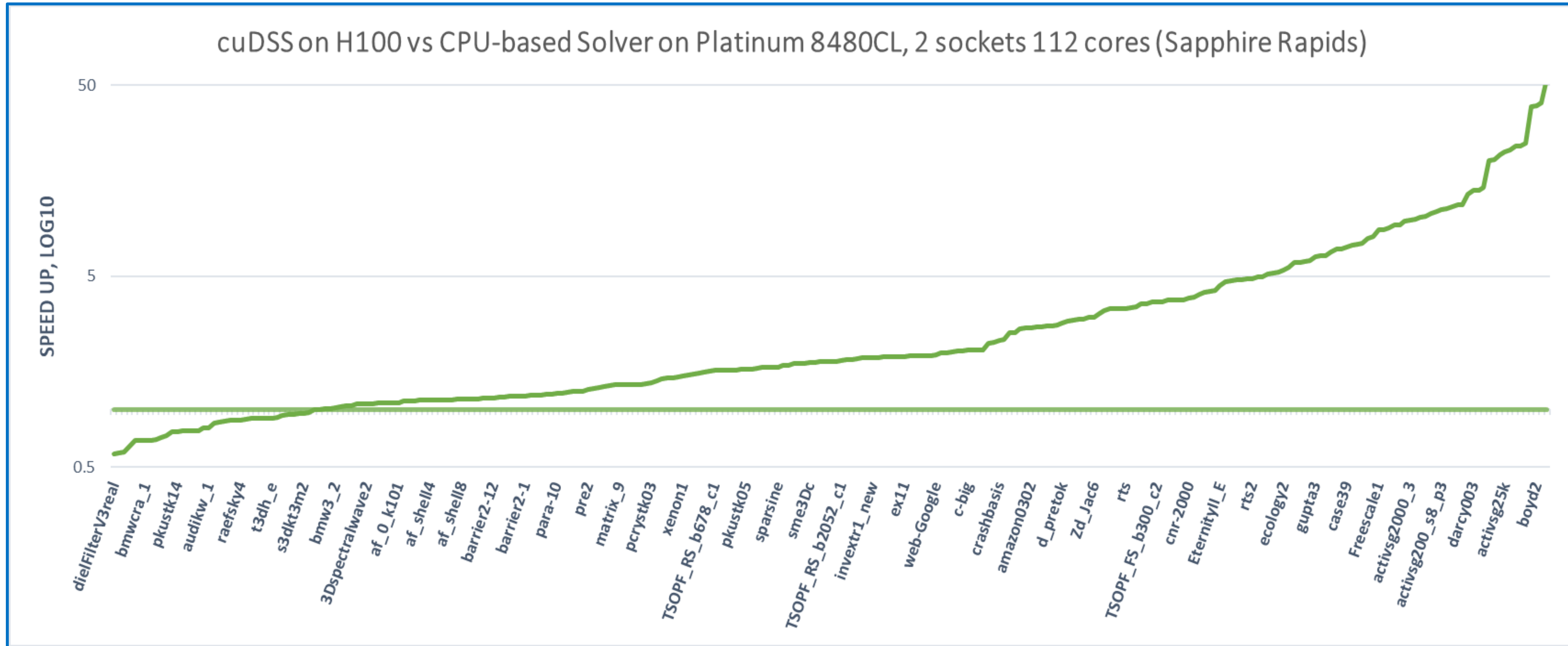
Perform forward and backward substitutions with triangular matrices L and U. Optionally it can include iterative refinement process using factorized matrix as a preconditioner.

NVIDIA CONFIDENTIAL. DO NOT DISTRIBUTE.

NVIDIA.

CUDA DIRECT SPARSE SOLVER

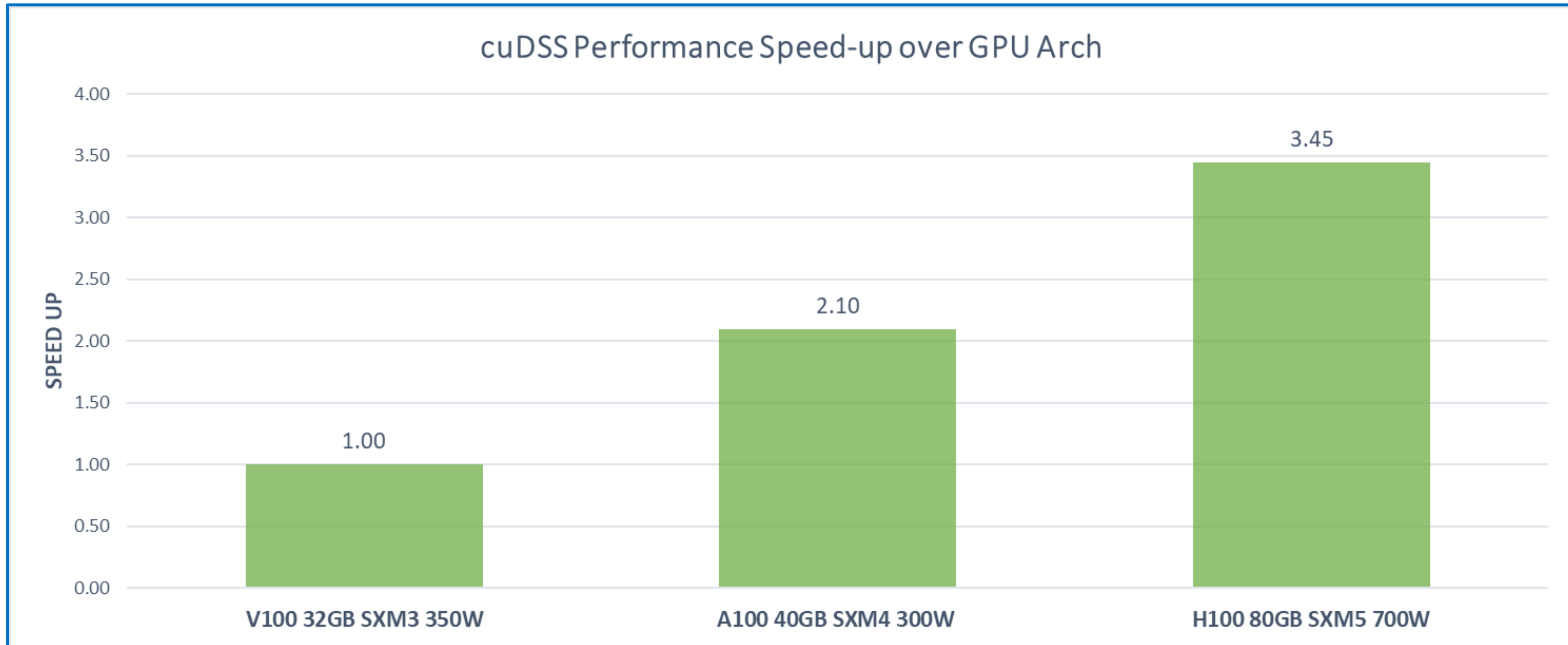
Performance overview



- 273 symmetric and non-symmetric matrices from Florida Collection (double precision)
- N from 5K to 4.6M, NNZ from 500K to 45M
- Performance summary
  - Reordering: Geomean = 1.2; Speed up: MAX = 5.5, MIN = 0.36; cuDSS faster in 53% of cases
  - Factorization: Geomean = 1.9; Speed up: MAX = 70, MIN = 0.52; cuDSS faster in 76% of cases
  - Solve: Geomean = 2.7; Speed up: MAX = 45, MIN = 0.41; cuDSS faster in 92% of cases

CUDA DIRECT SPARSE SOLVER

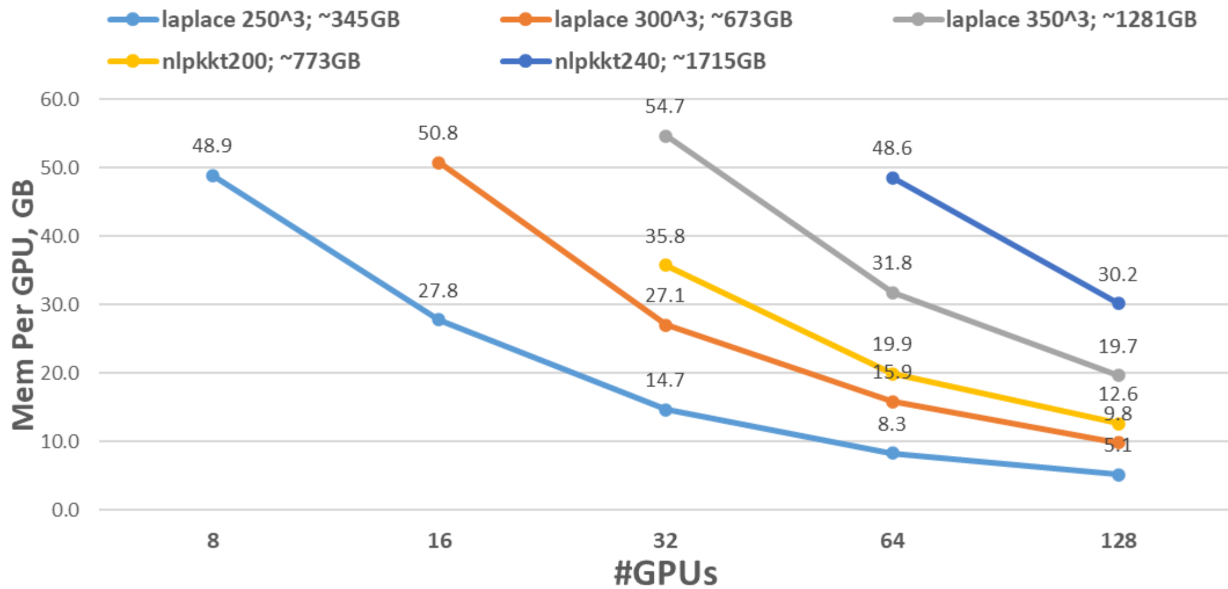
Performance overview

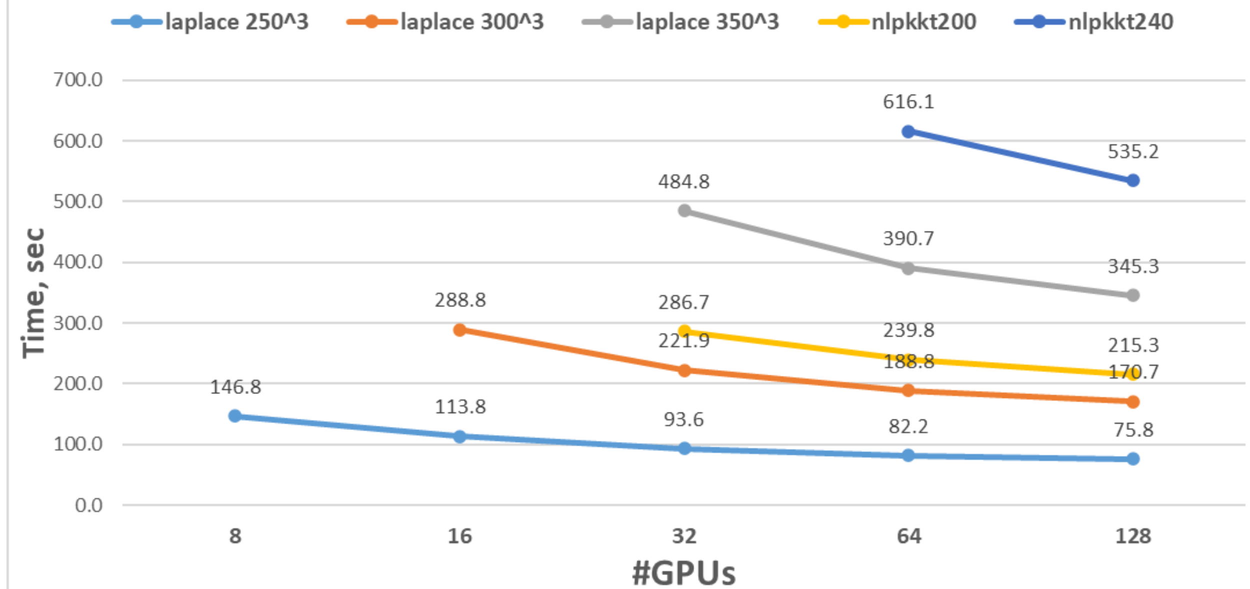


- A symmetric matrix from non-linear LP solver
- N = 6M, NNZ = 33M

CUDA DIRECT SPARSE SOLVER

Performance overview





- Symmetric indefinite matrices (double precision)
- Solve up to N = 65M and ~2200GB for L matrix. ~40GB per GPU (128 GPUs in total)
- Sequential reordering (MetisND)
- Distributed symbolic factorization, numeric factorization and solve

NVIDIA CONFIDENTIAL. DO NOT DISTRIBUTE.

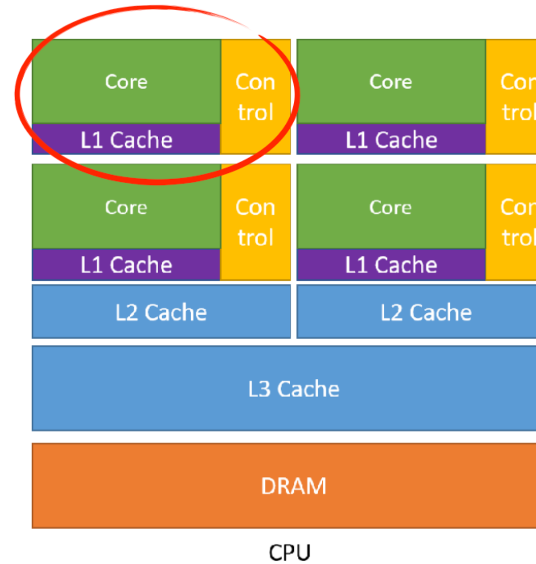
SCALING BEYOND AN SM

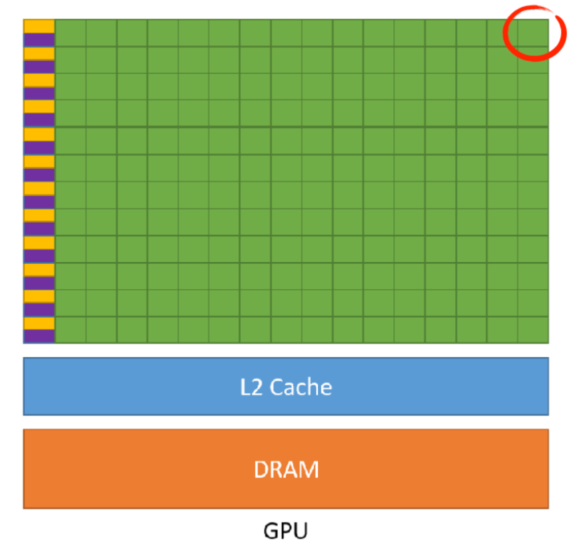
Eliminating kernel launch overhead

$$\begin{aligned}
&\textbf{for } k = 1{:}n-1 \\
&\quad \text{Determine } \mu \text{ with } k \le \mu \le n \text{ so } |A(\mu,k)| = \| A(k{:}n,k) \|_\infty \\
&\quad piv(k) = \mu \\
&\quad A(k,:) \leftrightarrow A(\mu,:) \\
&\quad \textbf{if } A(k,k) \neq 0 \\
&\qquad \rho = k+1{:}n \\
&\qquad A(\rho,k) = A(\rho,k)/A(k,k) \\
&\qquad A(\rho,\rho) = A(\rho,\rho) - A(\rho,k)A(k,\rho) \\
&\quad \textbf{end} \\
&\textbf{end}
\end{aligned}$$

sync()

sync()



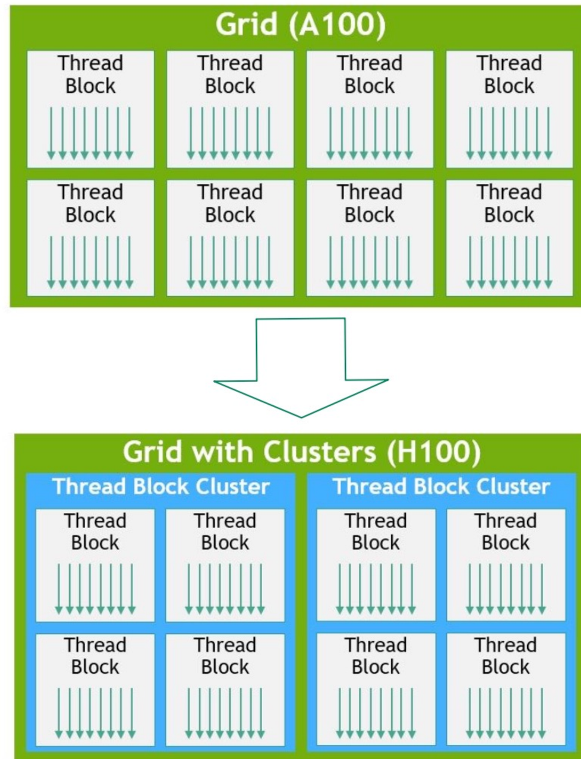


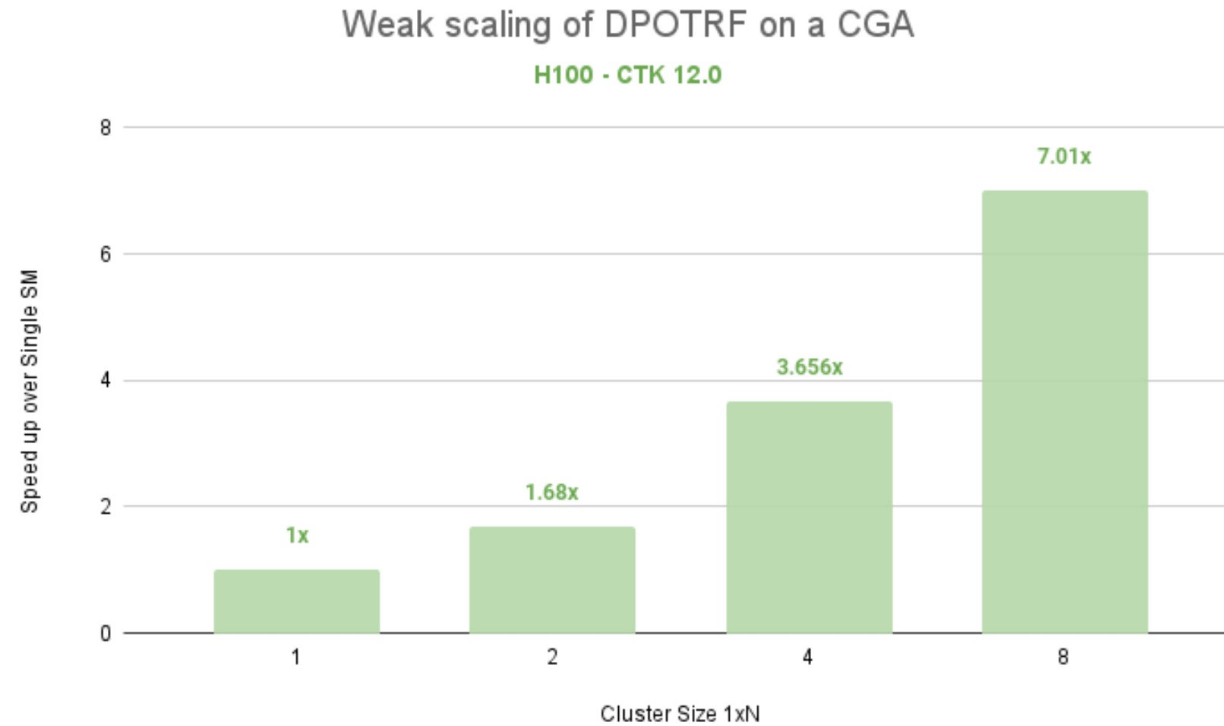
NVIDIA CONFIDENTIAL. DO NOT DISTRIBUTE.

NVIDIA.

SCALING BEYOND AN SM

Eliminating kernel launch overhead





INTER-CTA SYNCRHONIZATION

Scaling beyond *CGA*

- Producer-consumer implemented in GPU kernel.
    - Reduces kernel launch overhead; single kernel.
    - Further optimizations are possible. Example; prefetching data to shared memory while waiting, static sizes, etc

```
if (currenttask == 1) {
    /* have to wait for 1 dependencies */
    if ( threadIdx.x == 0 ) { do { ready = done[1]; } while ( ready != 1 ); }
    __syncthreads();
    ukernel_gemv ( CUBLAS_OP_N, 32, 32, (T_ELEM) -1.000000, &A[2 + 0*lda], lda, &x[0 + 0], 1, (T_ELEM) 1.000000, &x[2 + 0], 1);
    __syncthreads();

    /* Update dependencies for other CTAs. */
    if ( threadIdx.x == 0 ) {
    atomicAdd( (int*) &done[4], 1);
    };
}
```

NVIDIA CONFIDENTIAL. DO NOT DISTRIBUTE.

NVIDIA.

DEVICE EXTENSIONS LIBRARIES

Device-instantiated cuBLAS-like performance

```
// CUDA_CHECK_AND_EXIT - marco checks if function returns cudaSuccess; if not it prints
// the error code and exits the program
void introduction_example(value_type alpha, value_type *a, value_type *b, value_type beta, value_type *c) {
  constexpr auto t_mode = cublasdx::transpose_mode::non_transposed;
  using GEMM = decltype(Size<32, 32, 32>()
                + Precision<double>()
                + Type<type::real>()
                + TransposeMode<t_mode, t_mode>()
                + Function<function::MM>()
                + SM<700>()
                + Block()
                + BlockDim<256>());

  // Invokes kernel with GEMM::block_dim threads in CUDA block
  gemm_kernel<GEMM><<<1, GEMM::block_dim, GEMM::shared_memory_size>>>(alpha, a, b, beta, c);
```

NVIDIA CONFIDENTIAL. DO NOT DISTRIBUTE.

CUDA DIRECT SPARSE SOLVER

Future outlook

- GA release (1.0.0): later this year
- Functionality:
  - Matching + Scaling
  - Schur complement
  - QR factorization
  - Batch API
- Performance optimizations for customer use cases
- Python enablement
- Distribution: + pip wheels, conda

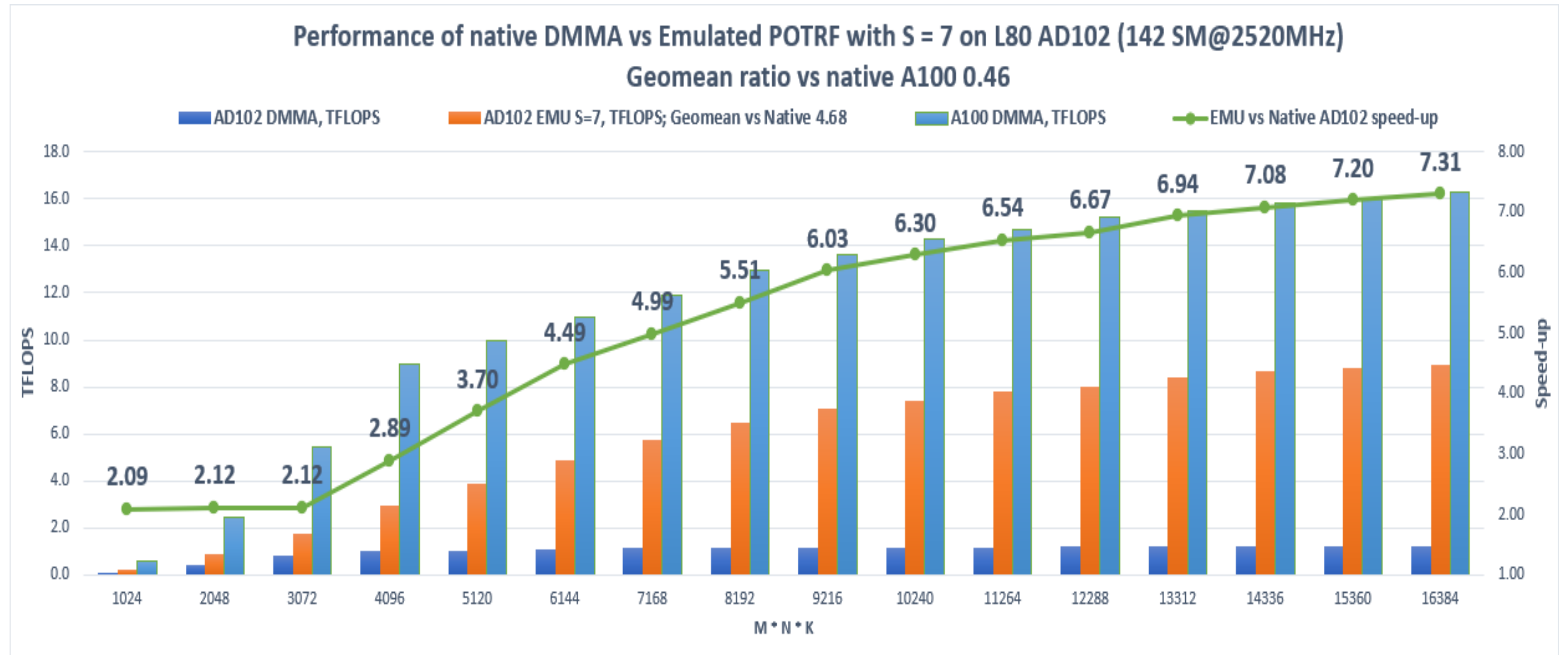
NVIDIA CONFIDENTIAL. DO NOT DISTRIBUTE.

NVIDIA

ACTIVE LINES OF RESEARCH

And opportunities for collaboration!

- FP64 emulation
- GPU-only fill-in reorderings
- SDDMM



NVIDIA CONFIDENTIAL. DO NOT DISTRIBUTE.

NVIDIA.

NVIDIA