GPU TECHNOLOGY CONFERENCE

GPU ACCELERATION OF WSMP
(WATSON SPARSE MATRIX PACKAGE)

NATALIA GIMELSHEIN	NVIDIA
ANSHUL GUPTA	IBM
STEVE RENNICH	NVIDIA
SEID KORIC	NCSA

WATSON SPARSE MATRIX PACKAGE (WSMP)

- Cholesky, LDL^T, LU factorization
- Distributed- and shared-memory algorithms
- Scalable, employs theoretically most scalable factorization algorithm
- Uses multifrontal method
- This work concentrates on accelerating numerical factorization phase

OBJECTIVE

- Acceleration of WSMP
  - Demonstrate suitability of GPUs for Sparse Direct Solves
  - Demonstrate suitability of GPUs for very irregular workloads
- Evaluate methods for accelerating WSMP
  - Simple methods can work well
  - More sophisticated methods can work better

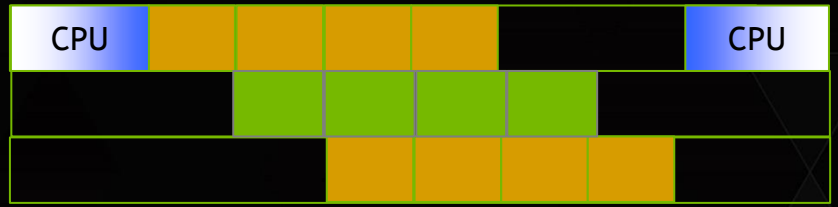
OUTLINE

- Minimally invasive acceleration of BLAS-intensive applications
- High level interface
- Acceleration techniques
- MPI acceleration

GPU ACCELERATION: MINIMALLY INVASIVE APPROACH

- Intercepting BLAS level 3 calls with big dimensions and sending them to GPU

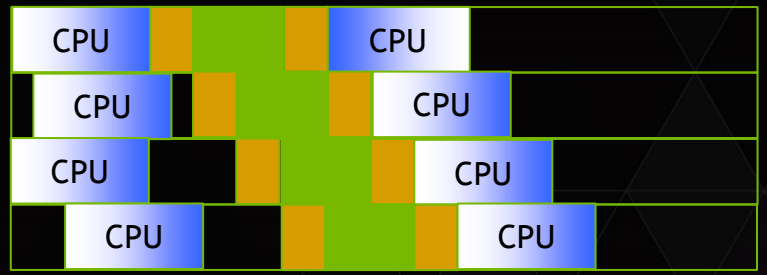
- Tiling to hide copies



- PCIe and host memory bandwidth are limiting
- But many of the matrices are not big enough to be tiled (sizes)

GPU ACCELERATION: MINIMALLY INVASIVE APPROACH

- We are intercepting simultaneous moderate-size calls
  - each one is not big enough to fully occupy GPU
- Use streams to
  - Increase GPU utilization
  - Hide copy-up and copy-down
  - Kernels may overlap



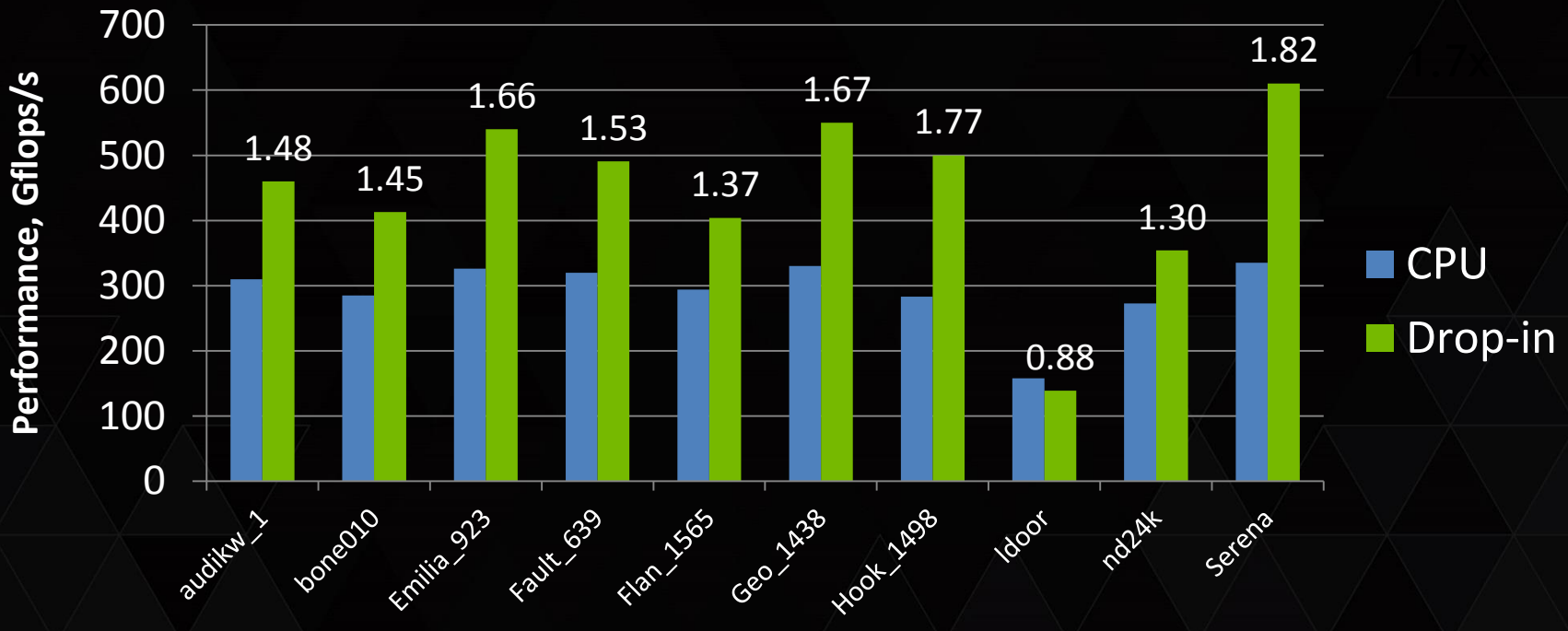
GPU ACCELERATION: MINIMALLY INVASIVE APPROACH

- Use host pinned buffers to increase copy-up/copy-down speed and enable asynchronous memory copies
- Send small BLAS calls to the CPU
  - (m,n<128, k<512)
- Large matrices are tiled individually
- Can be used with ANY application, not just WSMP
- No recompilation of the code required

RESULTS – SYSTEM USED

- Dual-socket Ivy-Bridge Xeon @ 3.0 Ghz
  - 20 cores total, PCIe gen3, E5-2690 v2
- Tesla K40, ECC on
- Intel MKL for host BLAS calls
- CUDA 6.5
- Standard cuBLAS routines are used for BLAS and BLAS-Like.

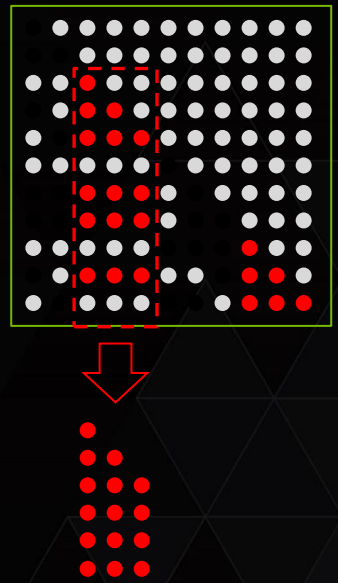
RESULTS - DROP-IN GPU ACCELERATION



2 x Xeon E5-2690 v2 + K40 (max boost, ECC=on)

SPARSE DIRECT SOLVERS

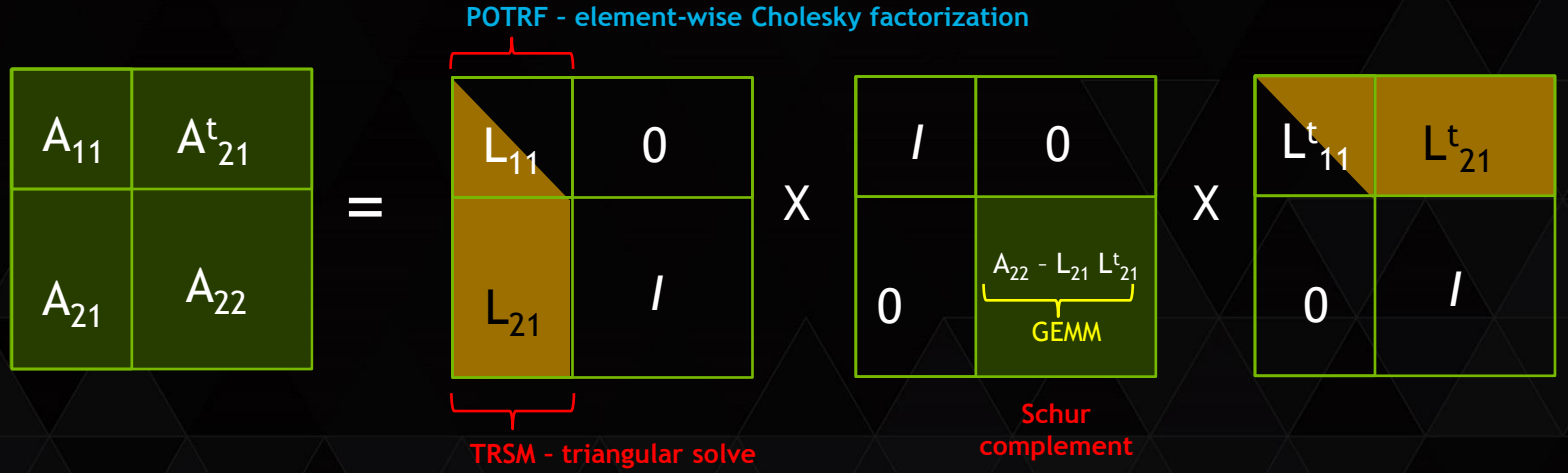
- Supernodes
  - collections of columns with similar non-zero pattern
  - provide opportunity for dense matrix math
  - grow with mesh size due to ‘fill’
  - The larger the model, the larger the supernodes
  - Supernodes in the factors are detected during symbolic factorization



DENSE BLOCK CHOLESKY

$$L_{11}\,L^{t}_{11} = A_{11} \quad \text{POTRF}$$
$$L_{21}\,L^{t}_{11} = A_{21} \quad \text{TRSM}$$
$$A^{*}_{22} = A_{22} - L_{21}\,L^{t}_{21} \quad \text{SYRK}$$

- Basis for sparse direct algorithms
  - **Emphasizes dense math**
  - Dominated by computation of Schur complement

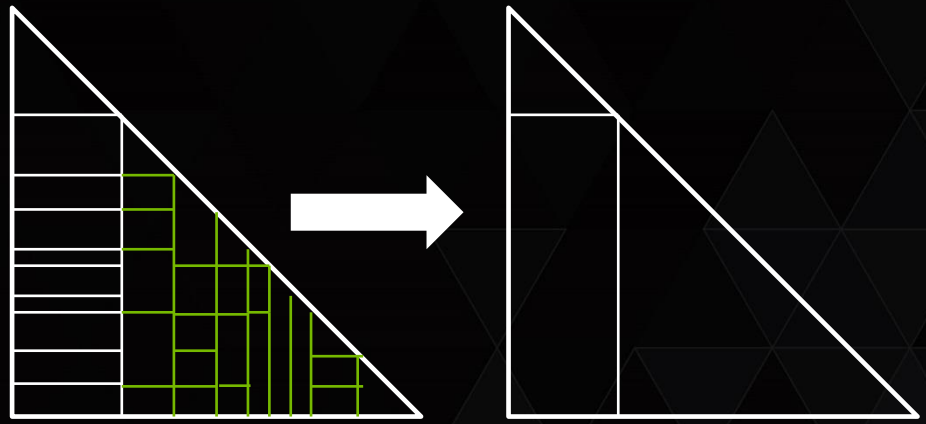


PARALLEL MULTIFRONTAL METHOD

- A task that owns the supernode
  - Assembles frontal matrix
  - Factors the supernode
  - Computes update frontal matrix, that will be used by the parent task
- In the beginning, many independent supernodes to be factored by parallel threads
- In the end, threads cooperate on the factorization of fewer remaining supernodes

POTENTIAL FOR IMPROVEMENT

- Potentially big BLAS calls are split into smaller ones, hurting performance
  - Same data is moved back and forth many times
  - A lot of pressure on PCIe for data movement
  - A lot of pressure on host memory bandwidth
- A few GB of memory allocated on the device and has to be pinned on the host

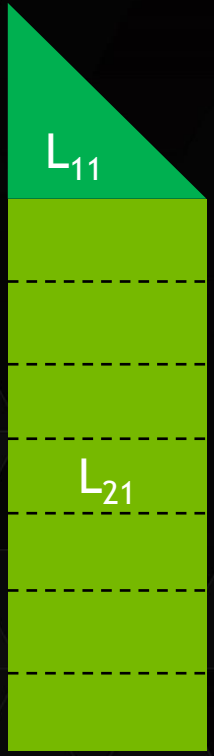


SOLUTION – HIGH LEVEL INTERFACE

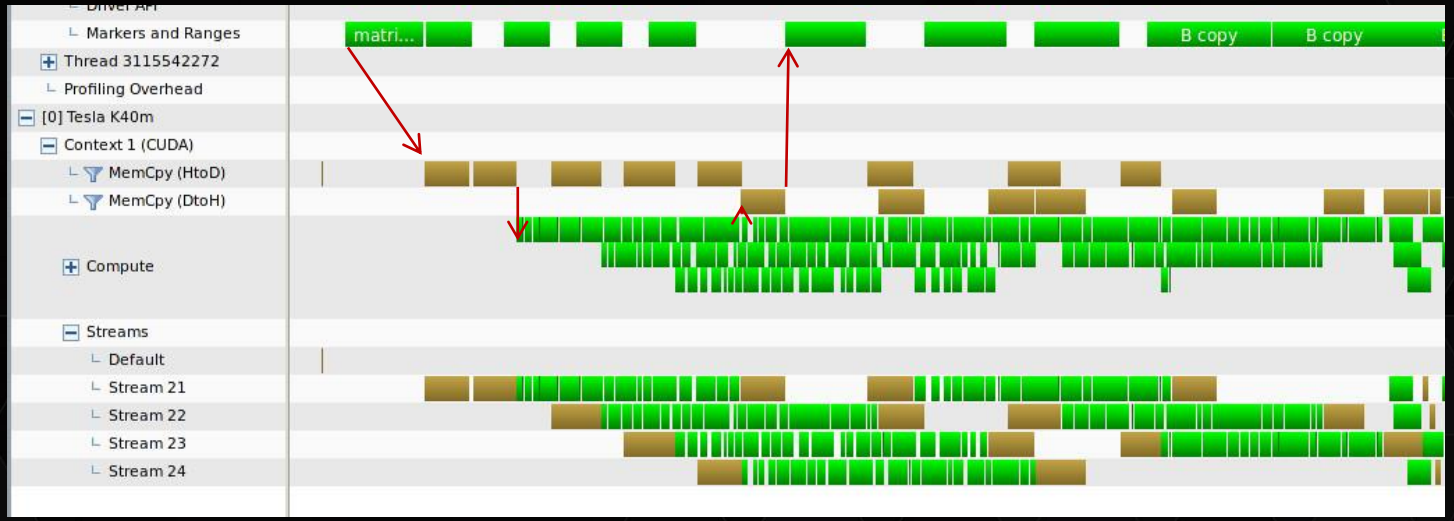
- Most of the work in Cholesky factorization is performed in dtrsmsyrk calls (dtrsm followed by dsyrk)
- Bigger BLAS dimensions are more favorable for the GPU
- Big dtrsmsyrk calls are sent to GPU (inner dimension >= 512)
- Still need to hide data transfer, but there is much less data motion between the host and the device
- Less memory needed on the device, less pinned memory on the host

DTRSM IMPLEMENTATION

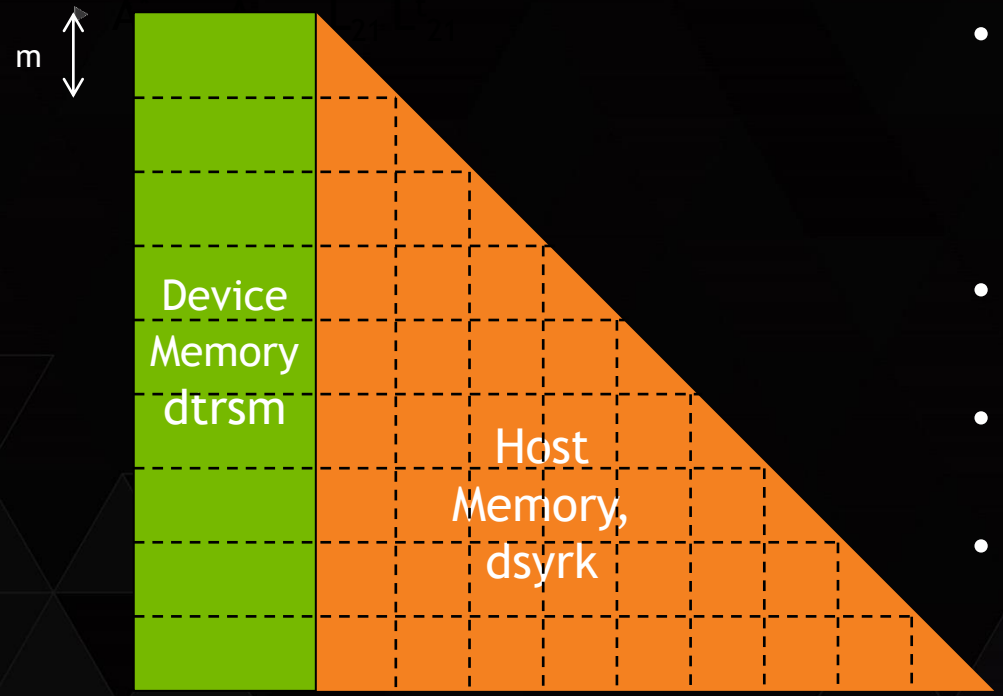
$L_{21}L_{11}^{T}=A_{21}$



- DTRSM is tiled
- Tiles and related copies are submitted to different streams
- Results are kept on the GPU, as they will be needed for dsyrk
- Copy is sent to the CPU
- Host buffers are used for staging host data

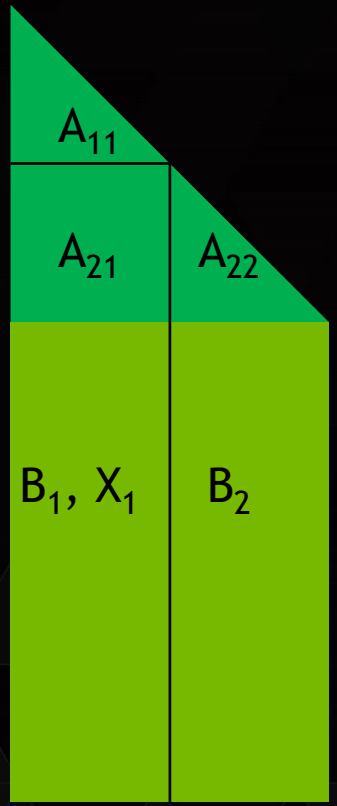


DSYRK IMPLEMENTATION



- Dsyrk is tiled
- Different tiles and related copies are submitted to the different streams
- $L_{21}L_{21}^t$ result is sent to host buffer
- On the host update A_{22}-$L_{21}L_{21}^t$ is performed
- m=384 with 4 streams enough to have GPU occupied

TILED DTRSM/DTRSMSYRK

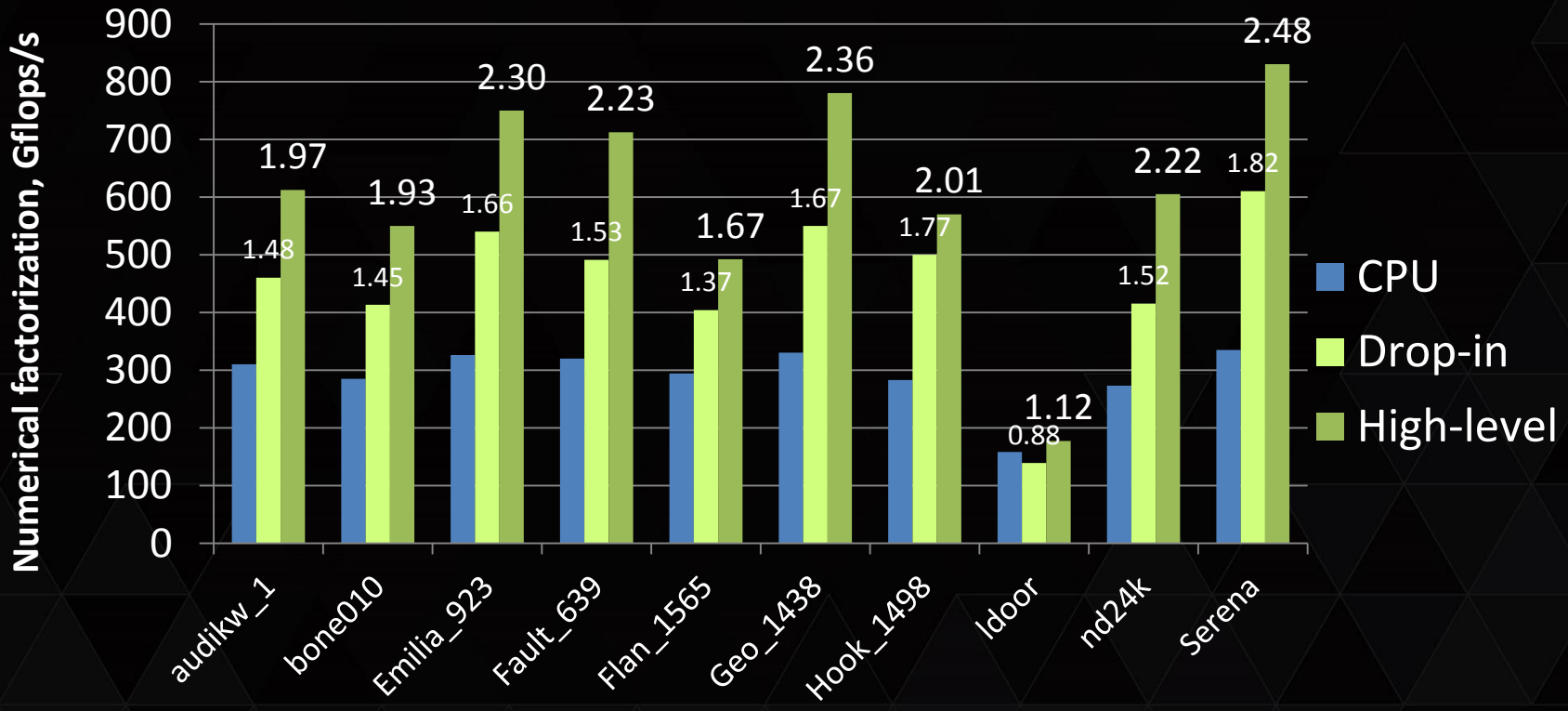


- When inner dimension of dtrsm is too big, it has to be tiled
- $X_1$is calculated from $X_1A_{11}^T=B_1$ as described before
- dsyrk update is performed if needed
- B_2 <- B_2-$X_1A_{21}^T$, X_1 is in the GPU memory, B_2 and A_{21} are on the CPU
- Process is repeated for the remainder of the matrix

MEMORY REQUIREMENTS

- For tiled dgemm with at most 2048 by 2048 quad-tiles:
  - 12 tiles, 400 MB
- For tiled dtrsm-syrk, with at most 2048 by 2048 tiles:
  - 12 tiles, 400 MB
- Buffer for dtrsm results:
  - With k <=2048 1 GB is enough for front size up to 61000, bigger fronts can be handled by successive calls to dtrsm and tiled dsyrk
- Total is less than 2 GB

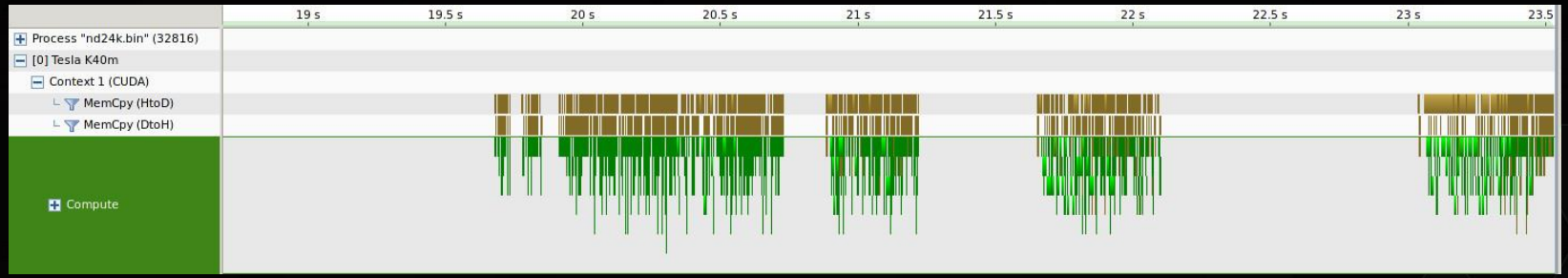
RESULTS



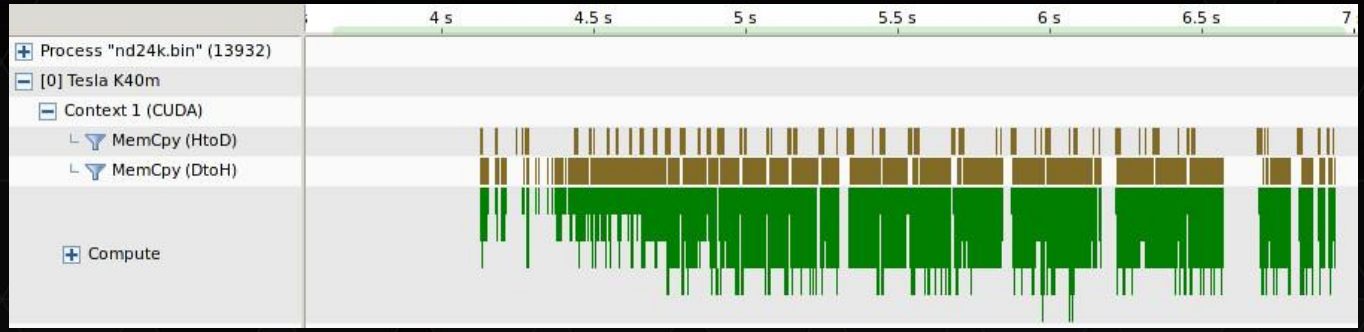
2 x Xeon E5-2690 v2 + K40 (max boost, ECC=on)

PROFILE

Drop-in



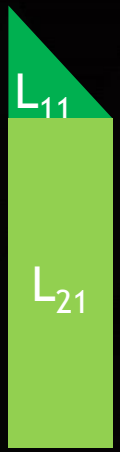
High-level API



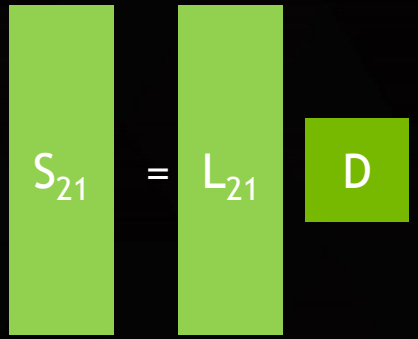
HIGH LEVEL INTERFACE FOR LDL FACTORIZATION

- WSMP signals what data is likely to be reused, and it is cached on the GPU

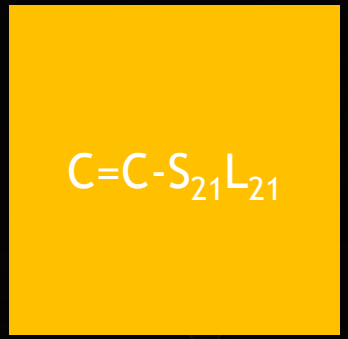
LDL^T FACTORIZATION

1) 

$L_{21}\ L^t{}_{11} = A_{21}$
dtrsm

2) 

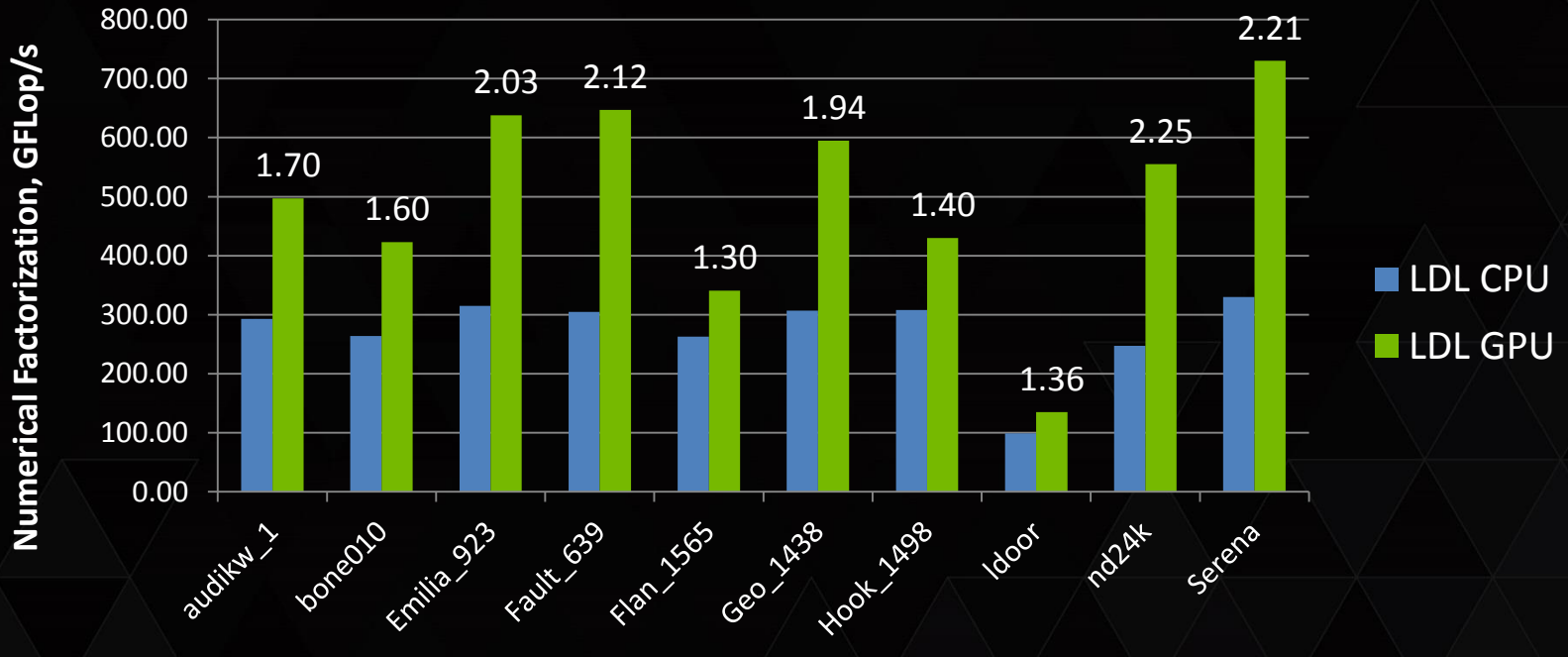
- scaling (cublasDdgmm)

3) 

- symmetric update (cublasDsyrkx)

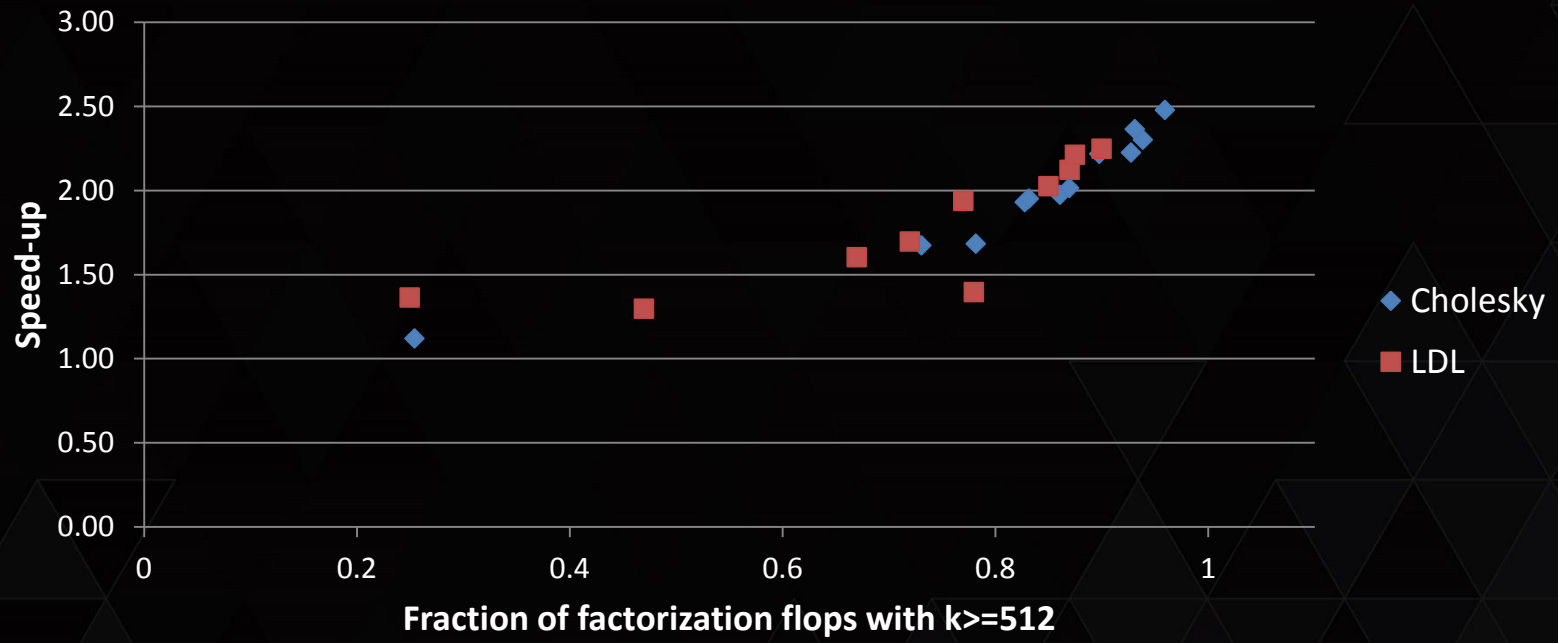
L_{21} and S_{21} are kept in the GPU memory between operations

LDL RESULTS



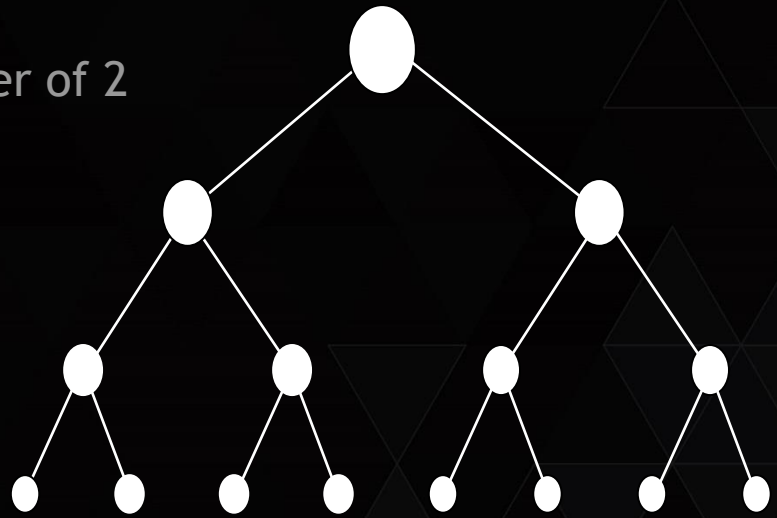
2 x Xeon E5-2690 v2 + K40 (max boost, ECC=on)

SPEEDUP



DISTRIBUTED MEMORY PARALLEL FACTORIZATION (MPI)

- Factorization by p processes
- Best performance when p is power of 2
- Factorization of levels below top log p levels is performed by the processes independently
- Processes cooperate on factoring upper levels of the elimination tree



DISTRIBUTED MEMORY PARALLEL FACTORIZATION

- Update matrices are distributed between processes working on them in a block-cyclic fashion
- Smaller block size provides better load balancing
- With bigger block size, efficiency of BLAS3 operations increases

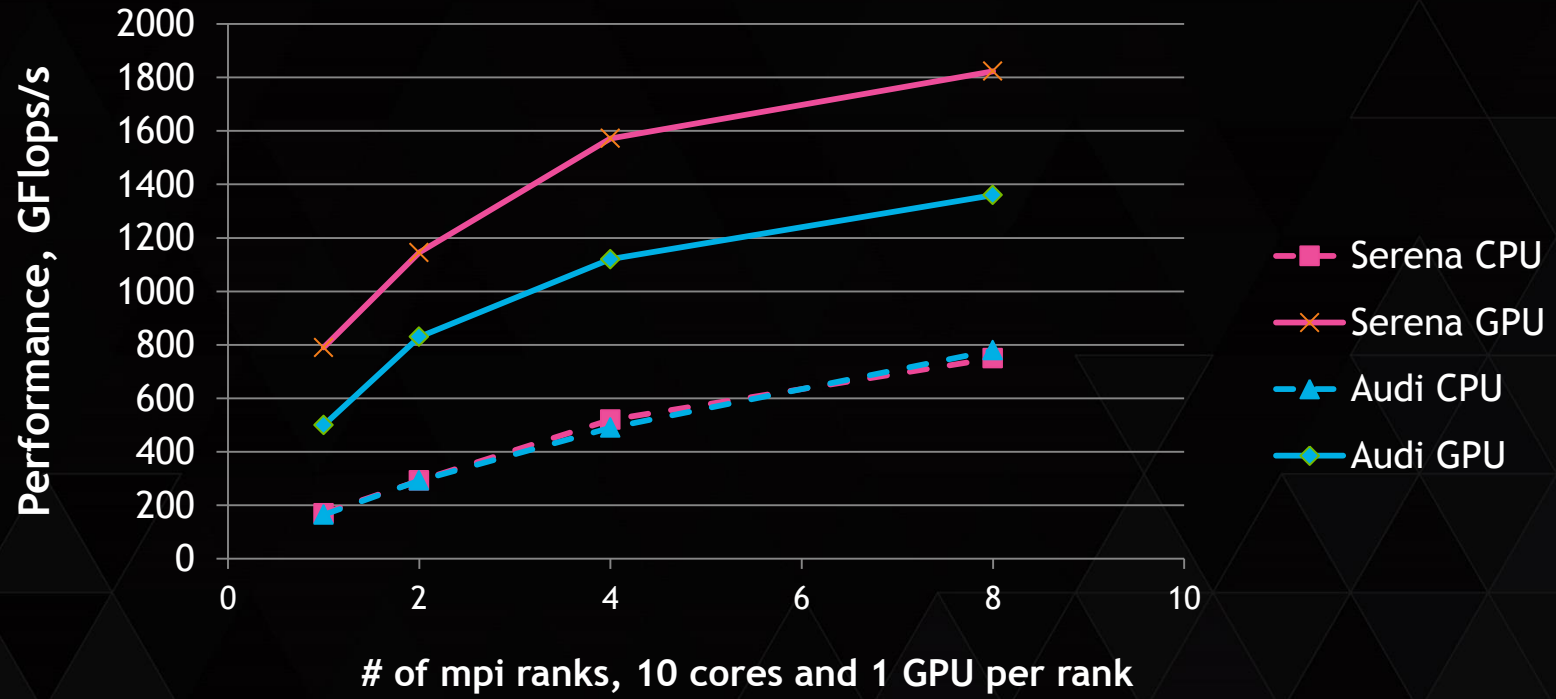
DISTRIBUTED MEMORY PARALLEL FACTORIZATION

- On the GPU block size ideally should be 512 or more
  - Determined by PCIe and host memcopy copy speed
- Large tiles hurt load balancing
- For our test system, 512 block size provides best performance

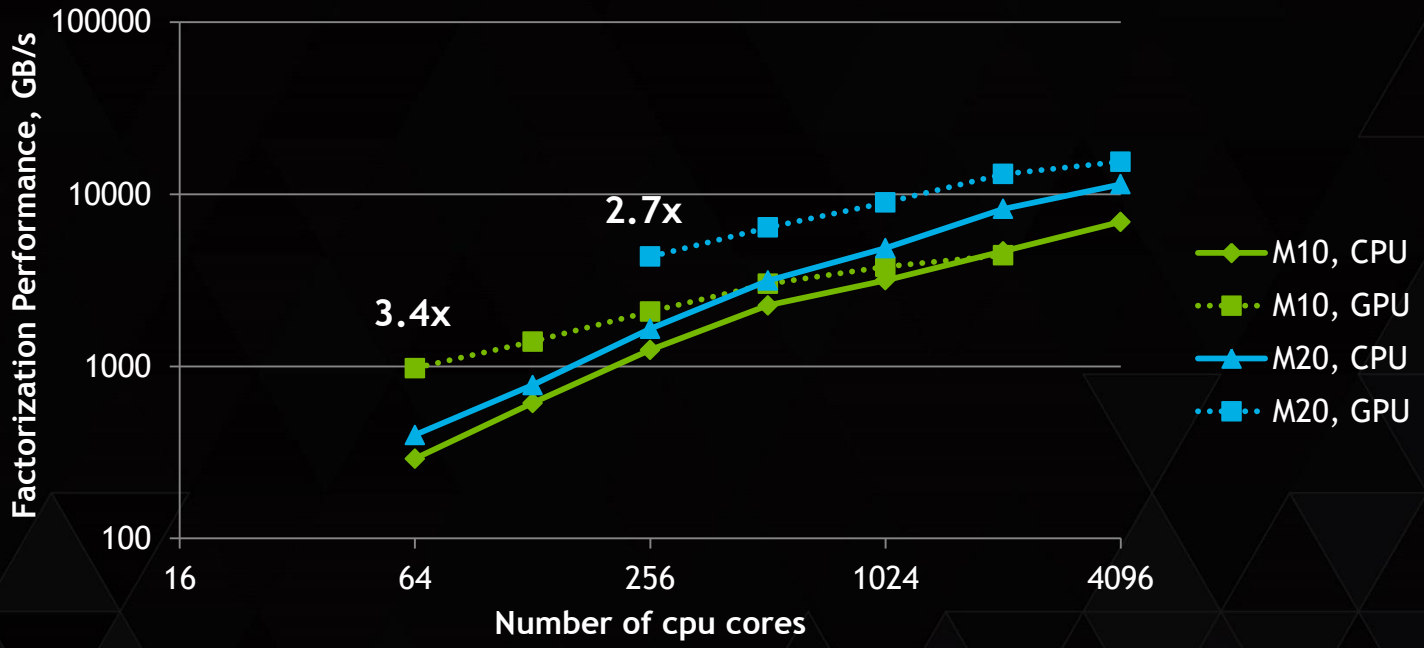
CACHING FOR MPI

- Dtrsm
  - results are kept in the GPU cache
- Dgemm
  - One of the matrices that is going to be reused is put in the GPU cache in the course of performing dgemm operation
  - One of the matrices are in the GPU cache, another matrix is not reused
- Dsyrk
  - Matrix can be in the GPU cache from previous operations, or on the host

MPI SCALING RESULTS



MPI SCALING ON BLUE WATERS



Blue Waters, AMD Interlagos, 16 CPU cores vs. 8 CPU cores + 1 K20 GPU

FUTURE WORK

- Share the work with CPU – now for many models it is idle as almost all work is sent to the GPU
- Multi-GPU
- Tuning to automatically set off-load cutoffs for a variety of systems

ACKNOWLEDGEMENTS

- The Private Sector Program at NCSA
- Blue Waters project, supported by NSF (award number OCI 07-25070) and the state of Illinois