TASK-BASED SPARSE DIRECT SOLVER FOR SYMMETRIC INDEFINITE SYSTEMS

Iain S. Duff and **Florent Lopez**

SIAM CSE19, Spokane, WA, 2019

Rutherford Appleton Laboratory

Objectives

Objective: solve $Ax = b$

Where A is a large, sparse and symmetrically-structured matrix

The algorithm needs to be:

Objectives

Objective: solve $Ax = b$

Where A is a large, sparse and symmetrically-structured matrix

The algorithm needs to be:

- Fast

Objectives

Objective: solve $Ax = b$

Where A is a large, sparse and symmetrically-structured matrix

The algorithm needs to be:

- Fast
- Numerically Stable

Objectives

Objective: solve $Ax = b$

Where A is a large, sparse and symmetrically-structured matrix

The algorithm needs to be:

- Fast
- Numerically Stable

In the context of the NLAFET project, we have developed the package SyLVER, that contains the two direct solvers:

Objectives

Objective: solve $Ax = b$

Where A is a large, sparse and symmetrically-structured matrix

The algorithm needs to be:

- Fast
- Numerically Stable

In the context of the NLAFET project, we have developed the package SyLVER, that contains the two direct solvers:

- SpLDLT: for symmetric systems:
  - positive-definite (LL^T)
  - indefinite (LDL^T)

Objectives

Objective: solve $Ax = b$

Where A is a large, sparse and symmetrically-structured matrix

The algorithm needs to be:

- Fast
- Numerically Stable

In the context of the NLAFET project, we have developed the package SyLVER, that contains the two direct solvers:

- SpLDLT: for symmetric systems:
  - positive-definite (LL^T)
  - indefinite (LDL^T)
- SpLU: for unsymmetric (LU) systems

Heterogenous CPU-GPU systems

Heterogeneous CPU-GPU architectures have been increasingly popular in the HPC community over the past few years:

Heterogenous CPU-GPU systems

Heterogeneous CPU-GPU architectures have been increasingly popular in the HPC community over the past few years:

Top500[1]: #1 (Power9&V100), #2 (Power9&V100), #5 (Haswell&P100)

[1] https://www.top500.org/

Heterogenous CPU-GPU systems

Heterogeneous CPU-GPU architectures have been increasingly popular in the HPC community over the past few years:

Top500[1]: #1 (Power9&V100), #2 (Power9&V100), #5 (Haswell&P100)

GPU architectures:

- High performance peak (GFlop/sec, GFlop/watt)
  e.g. NVIDIA V100 GPU: 7.8 TFlop/s theoretical peak (FP64), Cholesky peak 5.7 TFlop/s (FP64)

[1] https://www.top500.org/

Heterogenous CPU-GPU systems

Heterogeneous CPU-GPU architectures have been increasingly popular in the HPC community over the past few years:

Top500[1]: #1 (Power9&V100), #2 (Power9&V100), #5 (Haswell&P100)

GPU architectures:

- High performance peak (GFlop/sec, GFlop/watt)
  e.g. NVIDIA V100 GPU: 7.8 TFlop/s theoretical peak (FP64), Cholesky peak 5.7 TFlop/s (FP64)
- Support for reduced precision arithmetic such as FP16
  e.g. Cholesky peak 46.0 TFlop/s (FP16 using Tensor Cores)

[1] https://www.top500.org/

Heterogenous CPU-GPU systems

Heterogeneous CPU-GPU architectures have been increasingly popular in the HPC community over the past few years:

Top500[1]: #1 (Power9&V100), #2 (Power9&V100), #5 (Haswell&P100)

GPU architectures:

- ▲ High performance peak (GFlop/sec, GFlop/watt)
  e.g. NVIDIA V100 GPU: 7.8 TFlop/s theoretical peak (FP64), Cholesky peak 5.7 TFlop/s (FP64)
- ▲ Support for reduced precision arithmetic such as FP16
  e.g. Cholesky peak 46.0 TFlop/s (FP16 using Tensor Cores)
- ▼ Limited memory available on the device (16GB on V100)

[1] https://www.top500.org/

Heterogenous CPU-GPU systems

Heterogeneous CPU-GPU architectures have been increasingly popular in the HPC community over the past few years:

Top500[1]: #1 (Power9&V100), #2 (Power9&V100), #5 (Haswell&P100)

GPU architectures:

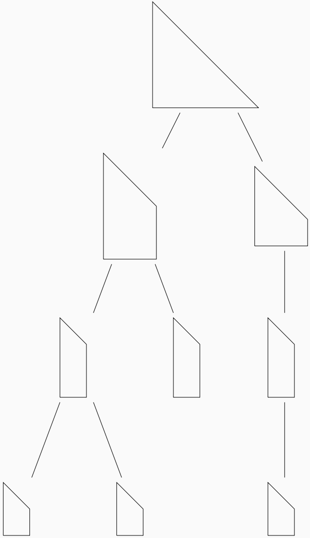
- ▲ High performance peak (GFlop/sec, GFlop/watt)
  e.g. NVIDIA V100 GPU: 7.8 TFlop/s theoretical peak (FP64), Cholesky peak 5.7 TFlop/s (FP64)
- ▲ Support for reduced precision arithmetic such as FP16
  e.g. Cholesky peak 46.0 TFlop/s (FP16 using Tensor Cores)
- ▼ Limited memory available on the device (16GB on V100)
- ▼ CPU $\leftrightarrow$ GPU memory transfer potentially slow
  e.g. PCIe 10$\times$ slower than NVLink but still common

[1] https://www.top500.org/

Sparse direct methods: parallelism

The multifrontal factorization is achieved with a topological traversal of the elimination tree:

1. Assemble the contributions from the descendant
2. Factor the column associated with the current node
3. Form the contribution blocks



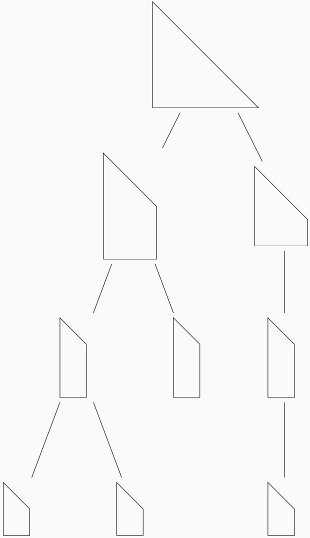
Sparse direct methods: parallelism

The multifrontal factorization is achieved with a topological traversal of the elimination tree:

1. Assemble the contributions from the descendant
2. Factor the column associated with the current node
3. Form the contribution blocks

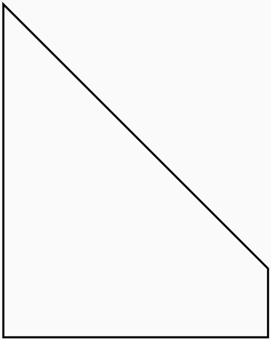
For the factorization and the solve phases, the sources of parallelism in the *elimination tree* include:

- Tree parallelism: nodes in independent branches can be processed concurrently
- Node parallelism: when a node is large enough, it may be processed in parallel

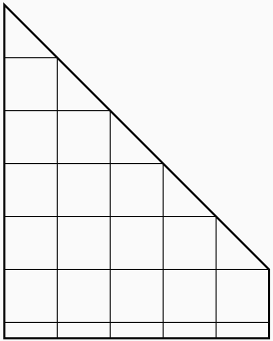


MULTIFRONTAL FACTORIZATION FOR MULTICORE CPUS

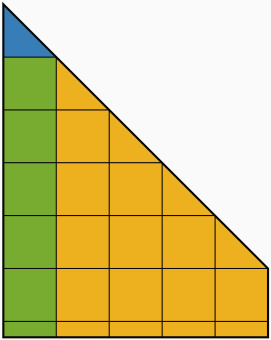
Dense Cholesky factorization on multicore CPUs



Dense Cholesky factorization on multicore CPUs



Dense Cholesky factorization on multicore CPUs

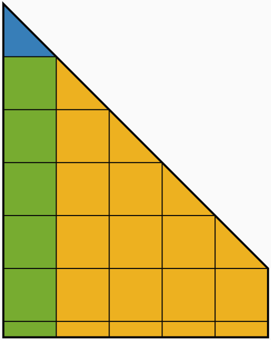


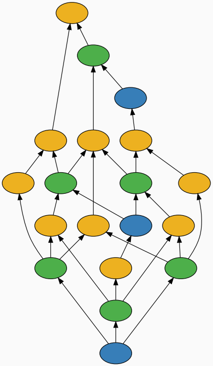
Factor diagonal block $A_{kk} = L_{kk} L_{kk}^T$

Apply pivots $L_{ik} = A_{ik} L_{kk}^{-T}$

Update trailing sub-matrix $A_{ij} -= L_{ik} L_{jk}^T$

Dense Cholesky factorization on multicore CPUs



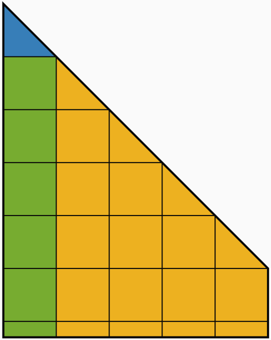


Factor diagonal block $A_{kk} = L_{kk} L_{kk}^T$

Apply pivots $L_{ik} = A_{ik} L_{kk}^{-T}$

Update trailing sub-matrix $A_{ij} -= L_{ik} L_{jk}^T$

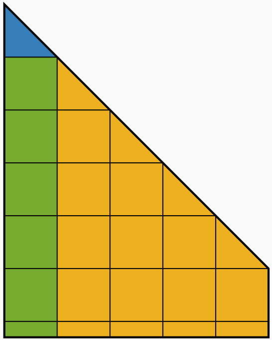
Dense Cholesky factorization on multicore CPUs



Indefinite systems: small diagonal elements and/or large off-diagonal elements might cause a lose of accuracy

Factor	diagonal block $A_{kk} = L_{kk} L_{kk}^T$
Apply	pivots $L_{ik} = A_{ik} L_{kk}^{-T}$
Update	trailing sub-matrix $A_{ij} -= L_{ik} L_{jk}^T$

Dense Cholesky factorization on multicore CPUs



Indefinite systems: small diagonal elements and/or large off-diagonal elements might cause a lose of accuracy

$\Longrightarrow$ avoid it by using pivoting

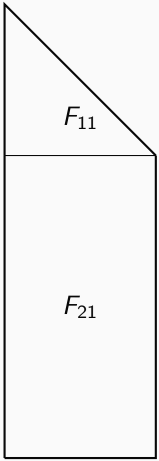
Factor diagonal block $A_{kk} = L_{kk} L_{kk}^T$

Apply pivots $L_{ik} = A_{ik} L_{kk}^{-T}$

Update trailing sub-matrix $A_{ij} -= L_{ik} L_{jk}^T$

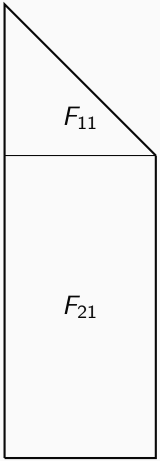
Pivoting strategies

Two main pivoting strategies for the LDL^T factorization:



Pivoting strategies

Two main pivoting strategies for the LDL^T factorization:

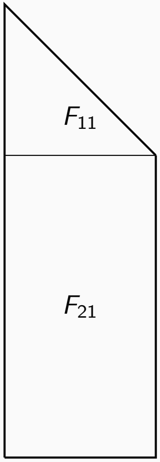


- Threshold Partial Pivoting (TPP)
  - ▲ Ensures that $|l_{ij}| < u^{-1}$ for some threshold u
  - ▼ Requires global communication on each columns
  - ○ Used in `HSL_MA97`

Numerically robust but hard to parallelize

Pivoting strategies

Two main pivoting strategies for the LDL^T factorization:



- Threshold Partial Pivoting (TPP)
  - ▲ Ensures that $|l_{ij}| < u^{-1}$ for some threshold u
  - ▼ Requires global communication on each columns
  - ○ Used in `HSL_MA97`

  Numerically robust but hard to parallelize

- Supernode Bunch-Kaufmann (SBK)
  - ▲ Performs pivoting in only within F_{11} using Bunch-Kaufmann algorithm
  - ▼ Requires the use of a pivots perturbation strategy
  - ○ Scaling and ordering techniques may be used to alleviate this problem
  - ○ Used in `PARDISO`

  Plenty of parallelism but potentially unstable

A Posteriori Threshold Pivoting

A Posteriori Threshold Pivoting (APTP) strategy: improve the parallelism of TPP while preserving numerical stability of the factorization

A Posteriori Threshold Pivoting

A Posteriori Threshold Pivoting (APTP) strategy: improve the parallelism of TPP while preserving numerical stability of the factorization

1. Speculative execution: execute some tasks speculatively, assuming no numerical issues occurred, then check for instability and backtrack if necessary:

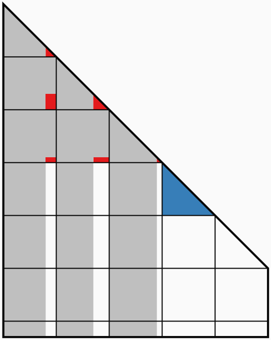
A Posteriori Threshold Pivoting

A Posteriori Threshold Pivoting (APTP) strategy: improve the parallelism of TPP while preserving numerical stability of the factorization

1. Speculative execution: execute some tasks speculatively, assuming no numerical issues occurred, then check for instability and backtrack if necessary:

2. Failed-in-place approach: keep the failed columns in place:

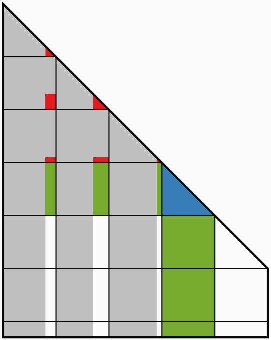
A Posteriori Threshold Pivoting



- Factor diagonal block using complete pivoting

```
do k = 1, nblk
  ! Factor diagonal block
  call Factor(A(k,k):RW, nelim_k)

end do
```

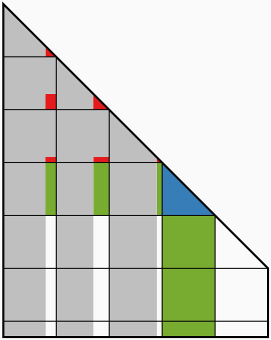
A Posteriori Threshold Pivoting



- Apply pivots in parallel using speculative execution

```
do k = 1, nblk
  ! Factor diagonal block
  call Factor(A(k,k):RW, nelim_k)
  ! Compute factors sub-diagonal on blocks
  do j = k+1, mblk
    call ApplyN(A(k,j):RW, A(k,k):R, nelim_k)
  end do
  ! Compute factors left-diagonal on blocks
  do j = 1, k-1
    call ApplyT(A(k,j):RW, A(k,k):R, nelim_k)
  end do









end do
```

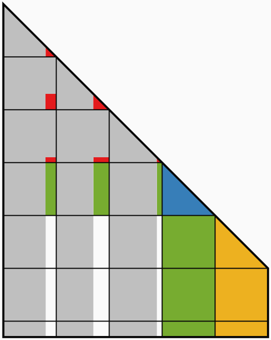
A Posteriori Threshold Pivoting



- Determine the number of successfully eliminated pivots w.r.t threshold u^{-1}

```
do k = 1, nblk
  ! Factor diagonal block
  call Factor(A(k,k):RW, nelim_k)
  ! Compute factors sub-diagonal on blocks
  do j = k+1, mblk
    call ApplyN(A(k,j):RW, A(k,k):R, nelim_k)
  end do
  ! Compute factors left-diagonal on blocks
  do j = 1, k-1
    call ApplyT(A(k,j):RW, A(k,k):R, nelim_k)
  end do
  ! Compute nelim_k
  call Adjust(A(:,k):R, nelim_k)
end do
```

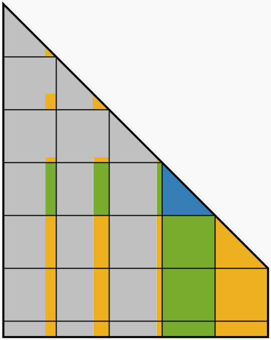
A Posteriori Threshold Pivoting



- Update blocks in the trailing sub-matrix

```
do k = 1, nblk
  ! Factor diagonal block
  call Factor(A(k,k):RW, nelim_k)
  ! Compute factors sub-diagonal on blocks
  do j = k+1, mblk
    call ApplyN(A(k,j):RW, A(k,k):R, nelim_k)
  end do
  ! Compute factors left-diagonal on blocks
  do j = 1, k-1
    call ApplyT(A(k,j):RW, A(k,k):R, nelim_k)
  end do
  ! Compute nelim_k
  call Adjust(A(:,k):R, nelim_k)
  ! Update trailing sub-matrix
  do j = k, nblk
    do i = j, mblk
      call UpdateNN(A(i,k):R, A(j,k):R, A(i,j):RW,
                    nelim_k)
    end do
  end do
end do
```

A Posteriori Threshold Pivoting



- Update uneliminated and failed entries in the left-diagonal blocks

```
do k = 1, nblk
  ! Factor diagonal block
  call Factor(A(k,k):RW, nelim_k)
  ! Compute factors sub-diagonal on blocks
  do j = k+1, mblk
    call ApplyN(A(k,j):RW, A(k,k):R, nelim_k)
  end do
  ! Compute factors left-diagonal on blocks
  do j = 1, k-1
    call ApplyT(A(k,j):RW, A(k,k):R, nelim_k)
  end do
  ! Compute nelim_k
  call Adjust(A(:,k):R, nelim_k)
  ! Update trailing sub-matrix
  do j = k, nblk
    do i = j, mblk
      call UpdateNN(A(i,k):R, A(j,k):R, A(i,j):RW,
                    nelim_k)
    end do
  end do
  ! Update uneliminated entries on left-diagonal
  ! blocks
  do j = 1, k-1
    do i = j, k-1
      call UpdateTT(A(k,i):R, A(k,j):R, A(i,j):RW,
                    nelim_k)
    end do
    do i = k, mblk
      call UpdateTT(A(i,k):R, A(k,j):R, A(i,j):RW,
                    nelim_k)
    end do
  end do
end do
```

A Posteriori Threshold Pivoting

A Posteriori Threshold Pivoting (APTP) strategy: improve the parallelism of TPP while preserving numerical stability of the factorization

1. Speculative execution: execute some tasks speculatively, assuming no numerical issues occurred, then check for instability and backtrack if necessary:

2. Failed-in-place approach: keep the failed columns in place:

A Posteriori Threshold Pivoting

A Posteriori Threshold Pivoting (APTP) strategy: improve the parallelism of TPP while preserving numerical stability of the factorization

1. Speculative execution: execute some tasks speculatively, assuming no numerical issues occurred, then check for instability and backtrack if necessary:
   - ▲ Increased parallelism
2. Failed-in-place approach: keep the failed columns in place:

A Posteriori Threshold Pivoting

A Posteriori Threshold Pivoting (APTP) strategy: improve the parallelism of TPP while preserving numerical stability of the factorization

1. Speculative execution: execute some tasks speculatively, assuming no numerical issues occurred, then check for instability and backtrack if necessary:
   - ▲ Increased parallelism
   - ▼ Backups of entries $\Longrightarrow$ increased memory footprint
2. Failed-in-place approach: keep the failed columns in place:

A Posteriori Threshold Pivoting

A Posteriori Threshold Pivoting (APTP) strategy: improve the parallelism of TPP while preserving numerical stability of the factorization

1. Speculative execution: execute some tasks speculatively, assuming no numerical issues occurred, then check for instability and backtrack if necessary:
   - ▲ Increased parallelism
   - ▼ Backups of entries $\implies$ increased memory footprint
2. Failed-in-place approach: keep the failed columns in place:
   - ▲ Dramatically reduce data movement

A Posteriori Threshold Pivoting

A Posteriori Threshold Pivoting (APTP) strategy: improve the parallelism of TPP while preserving numerical stability of the factorization

1. Speculative execution: execute some tasks speculatively, assuming no numerical issues occurred, then check for instability and backtrack if necessary:
   - ▲ Increased parallelism
   - ▼ Backups of entries $\Longrightarrow$ increased memory footprint
2. Failed-in-place approach: keep the failed columns in place:
   - ▲ Dramatically reduce data movement
   - ▼ Failed entries must be kept up-to-date $\Longrightarrow$ Reduced granularity

Parallel multifrontal factorization for multicore CPUs

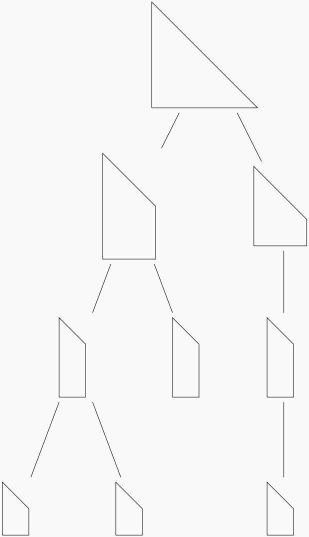
SpLDLT implements a multifrontal factorization using a Sequential Task Flow (STF) model

```
do n = 1, nnodes ! Topologically ordered
   ! Allocate memory
   call activate(front(n));
   ! Assemble fully-summned columns
   call assemble(front(n), children(n));
   ! Compute factors
   call factor(front(n));
   ! Assemble contribution block
   call assemble_contrib(front(n), children(n));
end do

! Wait for completion of submitted tasks
call task_wait_for_all();
```

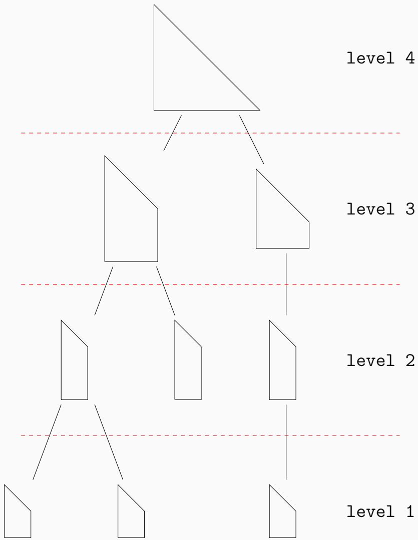
- `assemble`
  1. Allocate memory (fully-summed and contribution blocks)
  2. Assemble contributions from children nodes into fully-summed columns
- `factor` Factorize the fully-summed columns and form the contribution blocks
- `assemble_contrib` Assemble contributions from children nodes into contribution blocks

MULTIFRONTAL FACTORIZATION FOR GPU DEVICES

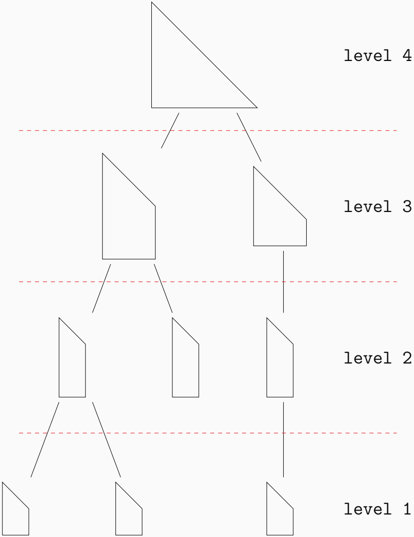
Multifrontal factorization for GPU devices



Multifrontal factorization for GPU devices

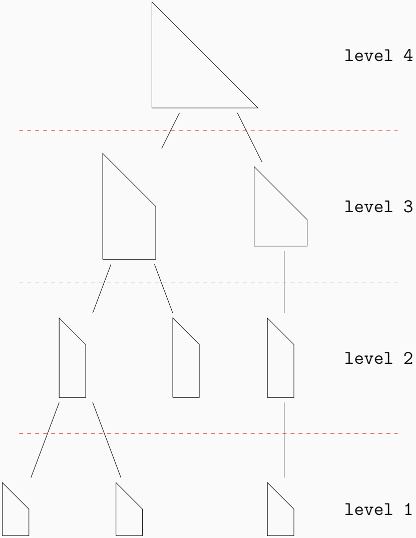


Multifrontal factorization for GPU devices




```
do lvl = 1, nlevels
   ! Allocate memory
   call activate_lvl(fronts(lvl))
   ! Assemble fully-summned columns
   call assemble_lvl(fronts(lvl), child(lvl))
   ! Compute factors
   call factor_lvl(fronts(lvl))
   ! Assemble contribution block
   call assemble_cb_lvl(fronts(lvl), child(lvl))
end do
```

Multifrontal factorization for GPU devices

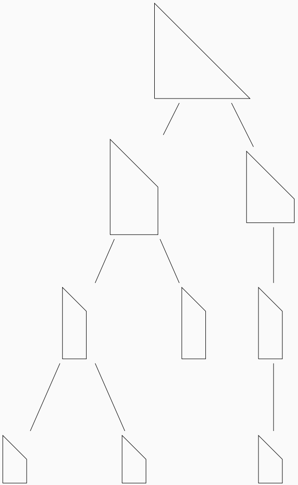



```
do lvl = 1, nlevels
   ! Allocate memory
   call activate_lvl(fronts(lvl))
   ! Assemble fully-summned columns
   call assemble_lvl(fronts(lvl), child(lvl))
   ! Compute factors
   call factor_lvl(fronts(lvl))
   ! Assemble contribution block
   call assemble_cb_lvl(fronts(lvl), child(lvl))
end do
```

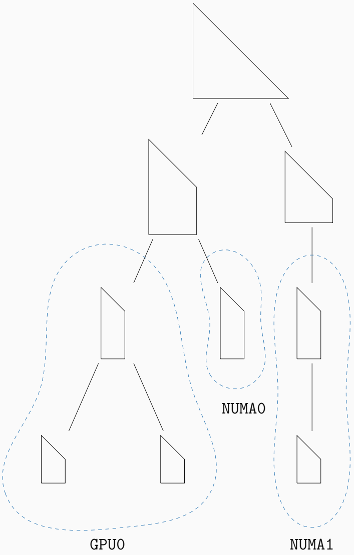
```
chol_gpu: do jj = 1, n(lvl), nb
   do kk = jj, jj+nb, ib
      ! Factor inner panel
      call potrf_gpu_batched(lvlnodes(kk))
      ! Update inner panel
      call syrk_gpu_batched(lvlnodes(kk))
      ! Update trailing submatrix
   end do
   call syrk_gpu_batched(lvlnodes(jj), &
        lvlnodes(jj+nb))
end do chol_gpu
```

HETEROGENEOUS CPU-GPU MULTIFRONTAL FACTORIZATION

Heterogeneous CPU-GPU multifrontal factorization

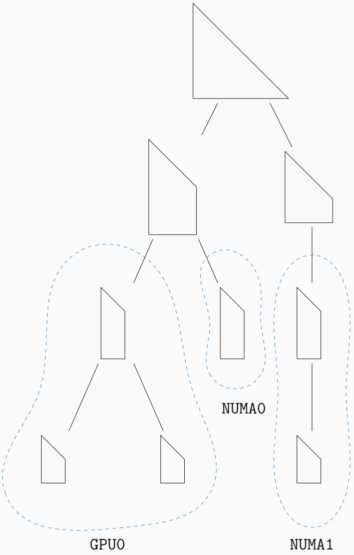


Heterogeneous CPU-GPU multifrontal factorization



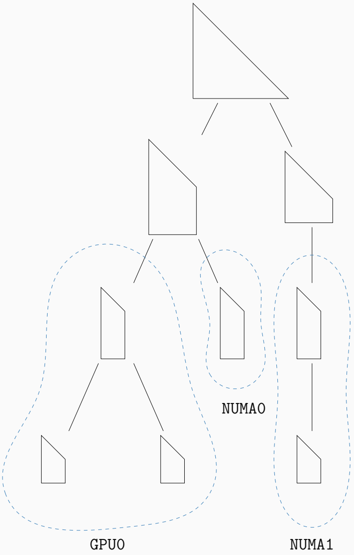
- Partition the assembly tree between NUMA regions and GPU devices

Heterogeneous CPU-GPU multifrontal factorization



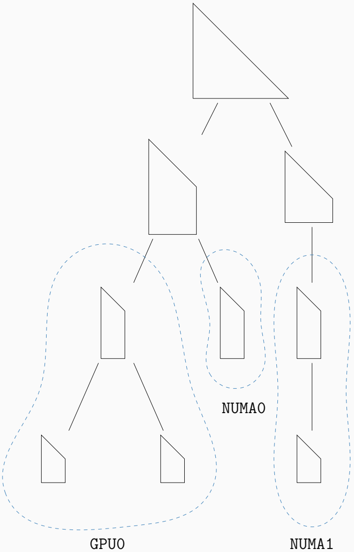
- Partition the assembly tree between NUMA regions and GPU devices
- balance $= \dfrac{\max_i(w_i \quad)}{\frac{1}{n_{\mathrm{res}}}\sum_j(w_j \quad)}$

Heterogeneous CPU-GPU multifrontal factorization



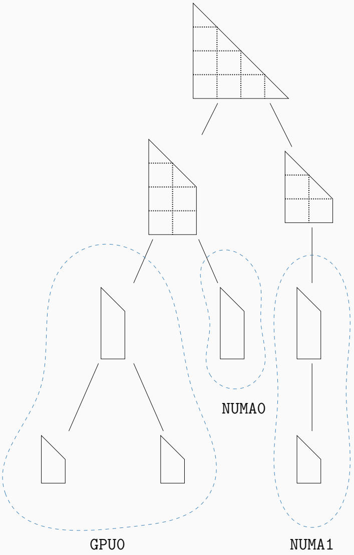
- Partition the assembly tree between NUMA regions and GPU devices
- balance $= \frac{\max_i (w_i/\alpha_i)}{\frac{1}{n_{\text{res}}}\sum_j (w_j/\alpha_j)}$
  - $\alpha_i = 1.0$ for NUMA regions
  - $\alpha_i = Perf_{GPU}/Perf_{CPU}$ for GPU devices

Heterogeneous CPU-GPU multifrontal factorization



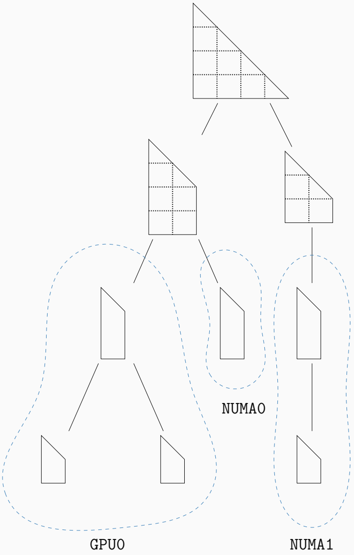
- Partition the assembly tree between NUMA regions and GPU devices
- balance $= \dfrac{\max_i(w_i/\alpha_i)}{\frac{1}{n_{\mathrm{res}}}\sum_j (w_j/\alpha_j)}$
  - $\alpha_i = 1.0$ for NUMA regions
  - $\alpha_i = Perf_{GPU}/Perf_{CPU}$ for GPU devices
- Within subtree:
  - NUMA partition: task-based factorization
  - GPU partition: batched factorization

Heterogeneous CPU-GPU multifrontal factorization



- Partition the assembly tree between NUMA regions and GPU devices
- balance $= \frac{\max_i(w_i/\alpha_i)}{\frac{1}{n_{\text{res}}}\sum_j(w_j/\alpha_j)}$
  - $\alpha_i = 1.0$ for NUMA regions
  - $\alpha_i = Perf_{GPU}/Perf_{CPU}$ for GPU devices
- Within subtree:
  - NUMA partition: task-based factorization
  - GPU partition: batched factorization
- Root partition: use task-based factorization where tasks run either on a CPU core or in a GPU stream

Heterogeneous CPU-GPU multifrontal factorization



- Partition the assembly tree between NUMA regions and GPU devices
- balance $= \dfrac{\max_i(w_i/\alpha_i)}{\frac{1}{n_{\text{res}}}\sum_j(w_j/\alpha_j)}$
  - $\alpha_i = 1.0$ for NUMA regions
  - $\alpha_i = Perf_{GPU}/Perf_{CPU}$ for GPU devices
- Within subtree:
  - NUMA partition: task-based factorization
  - GPU partition: batched factorization
- Root partition: use task-based factorization where tasks run either on a CPU core or in a GPU stream
- The DAG is dynamically scheduled in the root partition

Heterogeneous CPU-GPU multifrontal factorization

SpLDLT implementation details:

- We use StarPU to manage the DAG at the higher level

Heterogeneous CPU-GPU multifrontal factorization

SpLDLT implementation details:

- We use StarPU to manage the DAG at the higher level
- Two level parallelism in the DAG
  - Use OpenMP runtime system in NUMA regions as it comes with lower task management cost

Heterogeneous CPU-GPU multifrontal factorization

SpLDLT implementation details:

- We use StarPU to manage the DAG at the higher level
- Two level parallelism in the DAG
  - Use OpenMP runtime system in NUMA regions as it comes with lower task management cost
- The dynamic scheduling is handled by the Heteroprio
  - Task scheduling criterion: acceleration factor on given resources

Heterogeneous CPU-GPU multifrontal factorization

SpLDLT implementation details:

- We use StarPU to manage the DAG at the higher level
- Two level parallelism in the DAG
  - Use OpenMP runtime system in NUMA regions as it comes with lower task management cost
- The dynamic scheduling is handled by the Heteroprio
  - Task scheduling criterion: acceleration factor on given resources

▲ We are able to exploit GPU device while overcoming memory limitations

Heterogeneous CPU-GPU multifrontal factorization

SpLDLT implementation details:

- We use StarPU to manage the DAG at the higher level
- Two level parallelism in the DAG
  - Use OpenMP runtime system in NUMA regions as it comes with lower task management cost
- The dynamic scheduling is handled by the Heteroprio
  - Task scheduling criterion: acceleration factor on given resources

▲ We are able to exploit GPU device while overcoming memory limitations

▲ Exploitation of data locality within each subtree

Heterogeneous CPU-GPU multifrontal factorization

SpLDLT implementation details:

- We use StarPU to manage the DAG at the higher level
- Two level parallelism in the DAG
  - Use OpenMP runtime system in NUMA regions as it comes with lower task management cost
- The dynamic scheduling is handled by the Heteroprio
  - Task scheduling criterion: acceleration factor on given resources

▲ We are able to exploit GPU device while overcoming memory limitations

▲ Exploitation of data locality within each subtree

▲ Reduce the impact of the cost for the CPU $\leftrightarrow$ GPU memory transfers

NUMERICAL EXPERIMENTS

Numerical experiments

Experimental setup:

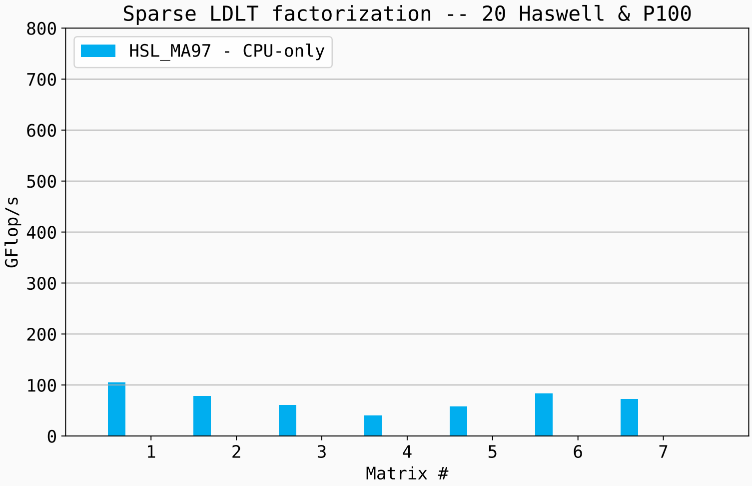
- 20 cores Intel Haswell machine (2 x E5-2695 v3)
- 1 NVIDIA Pascal P100
- 64 GB of RAM
- GNU compilers 7.1.0 with flags "-g -O2 -march=native"
- Intel MKL BLAS 11.3.1
- Metis 4.0.3

Numerical experiments: Sparse LDL^T factorization

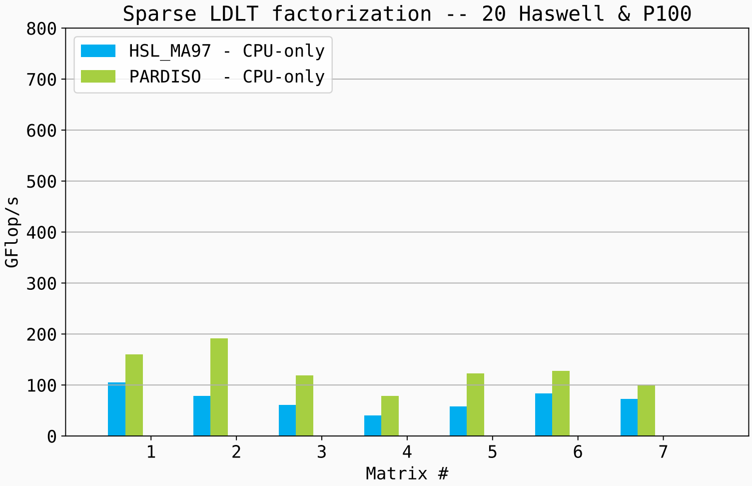
#	Problem	n $\times 10^3$	$nz(A)$ $\times 10^6$	$nz(L)$ $\times 10^6$	*flops* $\times 10^9$
1	Oberwolfach/t3dh	79.17	2.22	50.60	70.10
2	Lin/Lin	256.00	1.01	126.00	285.00
3	PARSEC/H2O	67.02	2.22	234.00	1290.00
4	GHS_indef/sparsine	50.00	0.80	207.00	1390.00
5	PARSEC/Ge99H100	112.98	4.28	669.00	7070.00
6	PARSEC/Ga10As10H30	113.08	3.11	690.00	7280.00
7	PARSEC/Ga19As19H42	133.12	4.51	823.00	9100.00

- HSL_MA97: multifrontal solver from the HSL library (CPU-only)
- PARDISO: supernodal solvers part of the MKL library (CPU-only)
- SpLDLT: our new solver, part of the SyLVER package (CPU-GPU)

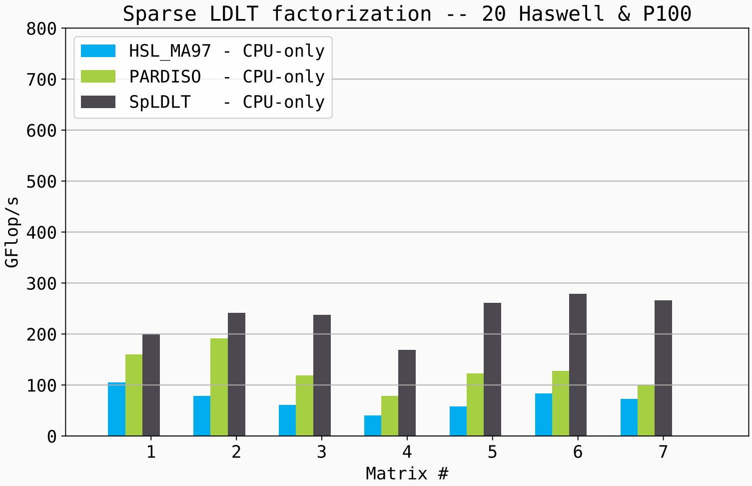
Numerical experiments: Sparse LDL^T factorization



Numerical experiments: Sparse LDL^T factorization

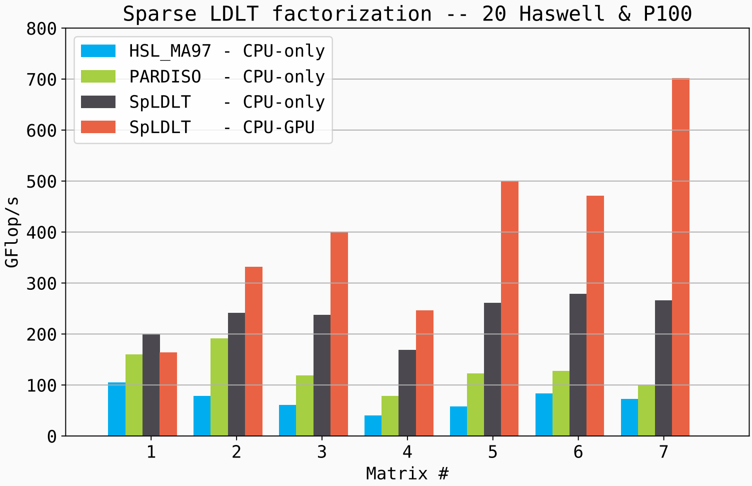


Numerical experiments: Sparse LDL^T factorization



- CPU-only SpLDLT code is up to $\times 3$ faster compared to PARDISO

Numerical experiments: Sparse LDL^T factorization



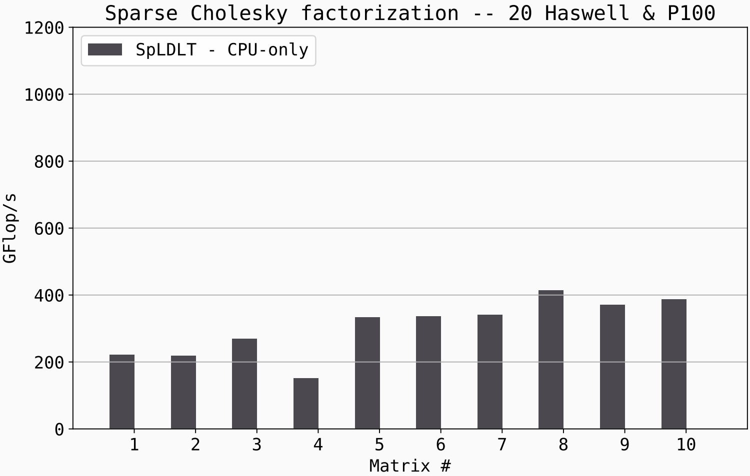
- CPU-only SpLDLT code is up to $\times 3$ faster compared to PARDISO
- Heterogeneous CPU-GPU SpLDLT code is up to $\times 3$ faster compared to CPU-only version

Numerical experiments: Sparse Cholesky factorization

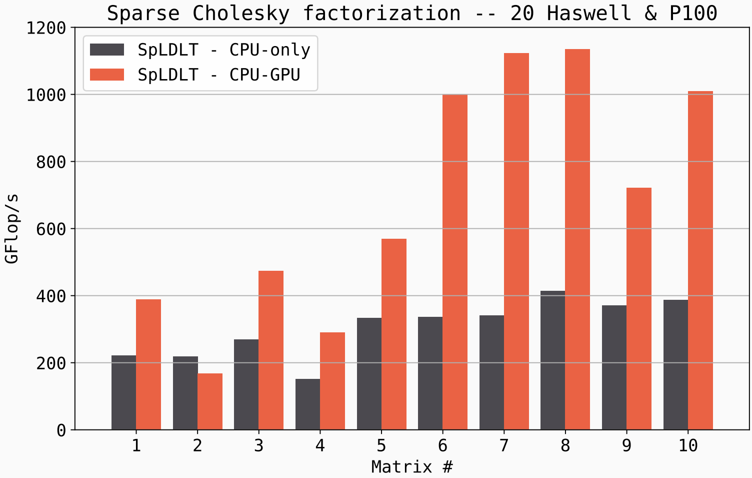
#	Problem	n $\times 10^3$	$nz(A)$ $\times 10^6$	$nz(L)$ $\times 10^6$	$flops$ $\times 10^9$
1	Koutsovasilis/F1	344	13.6	173.7	218.8
2	Oberwolfach/boneS10	915	28.2	278.0	281.6
3	ND/nd12k	36.0	7.1	116.5	505.0
4	ND/nd24k	72.0	14.4	321.6	2054.4
5	Janna/Flan_1565	1565	59.5	1477.9	3859.8
6	Oberwolfach/bone010	987	36.3	1076.4	3876.2
7	GHS_psdef/audikw_1	944	39.3	1242.3	5804.1
8	Janna/Fault_639	639	14.6	1144.7	8283.9
9	Janna/Hook_1498	1498	31.2	1532.9	8891.3
10	Janna/Emilia_923	923	21.0	1729.9	13661.1

- SpLDLT: our new solver, part of the SyLVERpackage (CPU-GPU)

Numerical experiments: Sparse Cholesky factorization



Numerical experiments: Sparse Cholesky factorization



Conclusions and future work

- Our sparse LDL^T factorization with APTP on multicore compares favourably with the state of the art solvers
- Using StarPU allows us GPU devices along with the multicores
- SpLDLT offers better numerical robustness compared to SBK like strategies

- We adapted the APTP pivoting to LU factorization..
- ..integrating it within SpLU using the same tree partitioning strategy as for SpLDLT

Thanks for listening!

Questions?