Toward a supernodal sparse direct solver over DAG runtimes

Sparse Days 2013, Toulouse

X. Lacoste
Guideline

Context and goals

Kernels
- Panel factorization
- Trailing supernodes update (CPU version)
- Sparse GEMM on GPU

Runtime

Results on DAG runtimes
- Matrices and Machines
- Multicore results
- GPU results

Improvement on granularity
- Smarter panel splitting

Results about granularity

Conclusion and future works
1

Context and goals
Context and goals

- Robust and efficient $Ax = b$ resolution using direct factorization
  - $\rightarrow$ PaStiX direct solver
- Factorization is time consuming, good performance required
- Emerging machines with many-cores and multiple GPUs
  - $\rightarrow$ use it all!
Possible solutions

- Many-cores: PaStiX already finely tuned to using MPI and P-Threads;
- Multiple-GPU and many-cores, two solutions:
  - Manually handle GPUs:
    - lot of work;
    - heavy maintenance;
  - Use dedicated runtime:
    - May lose the performance obtained on many-core;
    - Easy to add new computing devices.

Elected solution, runtime:

- StarPU: RUNTIME – Inria Bordeaux Sud-Ouest;
- PARSEC: ICL – University of Tennessee, Knoxville.
Major steps for solving sparse linear systems

1. **Analysis**: matrix is preprocessed to improve its structural properties ($A'x' = b'$ with $A' = P_nPD_rAD_cQP^T$)
2. **Factorization**: matrix is factorized as $A = LU$, $LL^T$ or $LDL^T$
3. **Solve**: the solution $x$ is computed by means of forward and backward substitutions
2

Kernels
Panel factorization

- Factorization of the diagonal block (\(xxTRF\));
- \(TRSM\) on the extra-diagonal blocks (ie. solves \(X \times b_d = b_{i,i>d}\) – where \(b_d\) is the diagonal block).

Figure: Panel update
Trailing supernodes update

- One global GEMM in a temporary buffer;
- Scatter addition (many AXPY).

Figure: Panel update
Why a new kernel?

- A BLAS call $\implies$ a CUDA startup paid;
- Many AXPY calls $\implies$ loss of performance.

$\implies$ need a GPU kernel to compute all the updates from $P_1$ on $P_2$ at once.
Kernels Sparse GEMM on GPU

How ?

auto-tunning GEMM CUDA kernel

- Auto-tunning done by the framework ASTRA developed by Jakub Kurzak for MAGMA and inspired from ATLAS;
- Computes $C \leftarrow \alpha AX + \beta B$, $AX$ split into a 2D tiled grid;
- A block of threads computes each tile;
- Each thread computes several entries of the tile in the shared memory and subtract it from $C$ in the global memory.

Sparse GEMM cuda kernel

- Based on auto-tunning GEMM CUDA kernel;
- Added two arrays giving first and last line of each blocks of $P_1$ and $P_2$;
- Computes an offset used when adding to the global memory.
Sparse GEMM on GPU

Figure: Panel update on GPU

```
sparse_gemm_cuda( char TRANSA, char TRANSB, int m, int n, 
cuDoublerComplex alpha, 
const cuDoublerComplex *d_A, int lda, 
const cuDoublerComplex *d_B, int ldb, 
cuDoublerComplex beta, 
cuDoublerComplex *d_C, int ldc, 
int blocknbr, const int *blocktab, 
int fblocknbr, const int *fblocktab, 
CUstream stream );
```

```c
blocknbr = 3;
blocktab = [ fr1,1, lr1,1, 
fr1,2, lr1,2, 
fr1,3, lr1,3 ];

fblocknbr = 2;
fblocktab = [ fr2,1, lr2,1, 
fr2,2, lr2,2 ];
```
GPU kernel experimentation

- \( Ncol_A = 100; \)
- \( Ncol_B = Nrow_{A_{11}} = 100; \)
- \( Nrow_A \) varies from 100 to 2000;
- Random number and size of blocks in \( A \);
- Random blocks in \( B \) matching \( A \);
- Get mean time of 10 runs for a fixed \( Nrow_A \) with different \( Nrow_A \) blocks distribution.

**Parameters**

**Figure:** GPU kernel experimentation
GPU kernel performance

Figure: Sparse kernel timing with 100 columns.
3
Runtime
Runtimes

- Task-based programming model;
- Tasks scheduled on computing units (CPUs, GPUs, ...);
- Data transfers management;
- Dynamically build models for kernels;
- Add new scheduling strategies with plugins;
- Get informations on idle times and load balances.
**Algorithm 1: StarPU tasks submission**

```plaintext
forall the Supernode \( S_1 \) do
    submit_panel \((S_1)\);
    /* update of the panel */
    forall the extra diagonal block \( B_i \) of \( S_1 \) do
        \( S_2 \leftarrow \) supernode_in_front_of \((B_i)\);
        submit_gemm \((S_1, S_2)\);
        /* sparse GEMM \( B_{k,k \geq i} \times B_i^T \) subtracted from \( S_2 \) */
    end
end
```
**PARSEC’s parametrized taskgraph**

\[
\text{panel}(j) \ [\text{high priority} = \text{on}]
\]

/* execution space */
\[
j = 0 \ldots \text{cblknbr}-1
\]

/* Extra parameters */
firstblock = \text{diagonal\_block\_of}(j)
lastblock = \text{last\_block\_of}(j)
lastbrow = \text{last\_brow\_of}(j) /* Last block generating an update on j */

/* Locality */
:A(j)
\[
\text{RW} \ A \leftarrow \text{leaf} ? \ A(j) : \text{C} \ \text{gemm}(\text{lastbrow})
\]
\[
\rightarrow A \ \text{gemm}(\text{firstblock}+1..\text{lastblock})
\]
\[
\rightarrow A(j)
\]

**Figure:** Panel factorization description in PARSEC
Giving more information to the runtime

**Definition of a new scheduler PaStiX work stealing**

- Use PaStiX static tasks placement;
- steal tasks from other contexts when no more tasks are ready (based on StarPU work stealing policy).

**Choose which GEMM will run on GPUs**

- statically decide to place only some panels on GPUs following a given criterium:
  - panel size;
  - number of update on the panel;
  - number of flops for the panel update.
- **PaRSEC** can place task on a given GPU whereas it’s more complicated with StarPU.
4

Results on DAG runtimes
Matrices and Machines

Matrices

<table>
<thead>
<tr>
<th>Name</th>
<th>N</th>
<th>NNZ_A</th>
<th>Fill ratio</th>
<th>OPC</th>
<th>Fact</th>
</tr>
</thead>
<tbody>
<tr>
<td>MHD</td>
<td>4.86×10^5</td>
<td>1.24×10^7</td>
<td>61.20</td>
<td>9.84×10^{12}</td>
<td>Float LU</td>
</tr>
<tr>
<td>Audi</td>
<td>9.44×10^5</td>
<td>3.93×10^7</td>
<td>31.28</td>
<td>5.23×10^{12}</td>
<td>Float LL^T</td>
</tr>
<tr>
<td>10M</td>
<td>1.04×10^7</td>
<td>8.91×10^7</td>
<td>75.66</td>
<td>1.72×10^{14}</td>
<td>Complex LDL^T</td>
</tr>
</tbody>
</table>

Machines

<table>
<thead>
<tr>
<th>Machine</th>
<th>Processors</th>
<th>Frequency</th>
<th>GPUs</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Romulus</td>
<td>AMD Opteron 6180 SE (4 × 12)</td>
<td>2.50 GHz</td>
<td>Tesla T20 (×2)</td>
<td>256 GiB</td>
</tr>
<tr>
<td>Mirage</td>
<td>Westmere Intel Xeon X5650 (2 × 6)</td>
<td>2.67 GHz</td>
<td>Tesla M2070 (×3)</td>
<td>36 GiB</td>
</tr>
<tr>
<td>Riri</td>
<td>Intel Xeon E7- 4870 (4 × 10)</td>
<td>2.40 GHz</td>
<td>None</td>
<td>1 TB</td>
</tr>
</tbody>
</table>
CPU only results on Audi

Figure: \( LL^T \) decomposition on Audi (double precision)
CPU only results on MHD

Figure: LU decomposition on MHD (double precision)
CPU only results on 10 Millions

Figure: $LDL^T$ decomposition on 10M (double complex)
GPU study on mirage

Figure: $LL^T$ decomposition on Audi (double precision)
5
Improvement on granularity
Improvements on granularity

- Graph preprocessing minimal blocking $\rightarrow$ reduce number of tasks;
- Smarter panel splitting to suppress low flop tasks.
Panel splitting

Why splitting panels?

- create more parallelism.

Drawback

- induce facing block splitting that can create many tiny blocks.

Solution

- smarter panel splitting;
- avoid tiny blocks creation which leads to inefficient BLAS.
A smarter split

- For each panel:
  - Construct a partition of the panel height with the number of facing blocks;
  - Decide to split where the number of splitted blocks is minimal.
A smarter split

Figure: Classical equal splitting
A smarter split

5 intervals partition:

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>facing blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>70</td>
<td>0</td>
</tr>
<tr>
<td>70</td>
<td>78</td>
<td>1</td>
</tr>
<tr>
<td>78</td>
<td>81</td>
<td>2</td>
</tr>
<tr>
<td>81</td>
<td>88</td>
<td>1</td>
</tr>
<tr>
<td>88</td>
<td>90</td>
<td>0</td>
</tr>
</tbody>
</table>

better to split only on of the two facing blocks, on row 78 or 81.
A smarter split

(a) Classical equal splitting

(b) Smarter adapted splitting
6
Results about granularity
Preprocessing option comparison on Audi, on Mirage

Smart panel splitting

- Factorization time reduction: 6-15%;
- Analyze time augmentation: 16-20%.

cmin 20

- Analyze time reduction: 80%;
- Less tasks may reduce runtime overhead, no effect on PaStiX factorization time.

Figure: $LL^T$ decomposition on Audi (double precision)
Results about granularity

Study on **Scotch** minimal subblock parameter (cmin), on Riri

![Figure: $LL^T$ decomposition on Audi (double precision)]
Conclusion and future works
Conclusion

▶ Timing and scaling close to \textsc{PaStiX};
▶ Speedup obtained with one (\textsc{StarPU}) or two (\textsc{PaRSEC}) GPUs and little number of cores;

Future works

▶ More locality :
  ▶ \textsc{StarPU} : use contexts to attach tasks to a pool of processing units;
  ▶ \textsc{PaRSEC} : Virtual processors : organize scheduling by socket;
▶ Streams : need streams to perform multiple kernel execution on a \textsc{GPU} at a time.
▶ Group tasks to reduce the runtime overhead (gather small tasks in \textsc{PaStiX} or let the runtime decide what is a small task);
▶ Distributed implementation (\textsc{MPI}), decide when to aggregate contribution or send \textsc{FANIN} or let the runtime decide.
Thanks !