Direct methods on GPU-based systems

Preliminary work towards a functioning code

Florent Lopez,
Joint work with IRIT Toulouse, LaBRI / Inria Bordeaux, LIP / Inria Lyon

Journée des doctorants 2012. Toulouse, September 7th
Context of the work
The multifrontal method

The multifrontal QR factorization is guided by a graph called elimination tree:

- at each node of the tree $k$ pivots are eliminated
- each node of the tree is associated with a relatively small dense matrix called frontal matrix (or, simply, front) which contains the $k$ columns related to the pivots and all the other coefficients concerned by their elimination
The tree is traversed in topological order (i.e., bottom-up) and, at each node, two operations are performed:
The Multifrontal method

The tree is traversed in topological order (i.e., bottom-up) and, at each node, two operations are performed:

- **assembly**: a set of coefficient from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are *stacked* to form the frontal matrix.

- **factorization**: the $k$ pivots are eliminated through a complete QR factorization of the frontal matrix. As a result we get:
  - $k$ rows of the global $R$ factor
  - a bunch of Householder vectors
  - a triangular contribution block that will be assembled into the father's front.
The Multifrontal method

The tree is traversed in **topological order** (i.e., bottom-up) and, at each node, two operations are performed:

- **assembly**: a set of coefficient from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are **stacked** to form the frontal matrix.

- **factorization**: the $k$ pivots are eliminated through a complete QR factorization of the frontal matrix. As a result we get:
  - $k$ rows of the global $R$ factor
  - a bunch of Householder vectors
  - a triangular *contribution block* that will be assembled into the father’s front.
GPUs may be used as powerful accelerators for HPC applications:

▲ High computational performance (comparison GPU-CPU: $10\times$ faster, memory access $5\times$ faster)
▲ Energy efficient

despite these capabilities, the use of GPUs is challenging:
▼ Complex architectures (comparison GPU-CPU: $100\times$ more cores)
▼ CPU-GPU programming models incompatible.
▼ CPU $\leftrightarrow$ GPU transfers are expensive (no shared memory)

⇒ specific algorithms
CPU-GPU hybrid architectures

An extremely heterogeneous workload
A heterogeneous architecture
mapping tasks is challenging
One option is to do the mapping by hand (see T. Davis' talk at SIAM PP12). This requires a very accurate performance models difficult to achieve.
Another option is to exploit the features of a modern runtime system capable of handling the scheduling and the data coherency in a dynamic way.
Runtime system: abstract layer between application and machine with the following features:

- Automatic detection of the task dependencies
- Dynamic task scheduling on different types of processing units.
- Management of *multi-versioned* tasks (an implementation for each type of processing unit)
- Coherency management of manipulated data.
Multifrontal QR factorization on multicore
Parallelism comes from two sources:

- **Tree**: nodes in separate branches can be treated independently
- **Node**: large nodes can be treated by multiple processes

In `qr_mumps` both sources are exploited consistently, by partitioning the frontal matrices and replacing the elimination tree with a DAG:
The multifrontal QR factorization: StarPU integration

- Depending on the input/output, StarPU detects the dependencies among tasks
- Depending on the availability of resources and the data placement, StarPU decides where to run a task
The easy way: replace all the

call `operation1(i1, ..., in, o1, ..., om)`

with

call `submit_task(operation1, i1, ..., in, o1, ..., om)`

and let StarPU do all the work
This is functionally correct but the DAG may have millions of nodes which makes the scheduling job too complex and memory consuming.
Our approach: We give to StarPU a limited view of the DAG; this is achieved by defining tasks that submit other tasks.
In the DAG we define

- **activation tasks**, i.e., tasks in charge of allocating the memory and preparing the data structures needed for processing a front.
• All the activation tasks are submitted at once with the right dependencies and very low priority. Each of them submits other tasks with higher priority.

• The runtime handles a DAG whose size is proportional only to the number of fronts that are active at a given moment.

• Tree traversal orders can be identified such that the size of this dynamic DAG is as small as possible but big enough to feed all the threads.
• All the activation tasks are submitted at once with the right dependencies and very low priority. Each of them submits other tasks with higher priority.

• The runtime handles a DAG whose size is proportional only to the number of fronts that are active at a given moment.

• Tree traversal orders can be identified such that the size of this dynamic DAG is as small as possible but big enough to feed all the threads.
• All the activation tasks are submitted at once with the right dependencies and very low priority. Each of them submits other tasks with higher priority.

• The runtime handles a DAG whose size is proportional only to the number of fronts that are active at a given moment.

• Tree traversal orders can be identified such that the size of this dynamic DAG is as small as possible but big enough to feed all the threads.
• All the activation tasks are submitted at once with the right dependencies and very low priority. Each of them submits other tasks with higher priority.

• The runtime handles a DAG whose size is proportional only to the number of fronts that are active at a given moment.

• Tree traversal orders can be identified such that the size of this dynamic DAG is as small as possible but big enough to feed all the threads.
• All the activation tasks are submitted at once with the right dependencies and very low priority. Each of them submits other tasks with higher priority.

• The runtime handles a DAG whose size is proportional only to the number of fronts that are active at a given moment.

• Tree traversal orders can be identified such that the size of this dynamic DAG is as small as possible but big enough to feed all the threads.
Experimental setup

- **Platform:**
  - 4× AMD hexacore
  - 76 GB of memory (in 4 NUMA modules)
  - GNU 4.4 compilers
  - MKL 10.2

- **Problems:** a set of matrices from the UF collection

<table>
<thead>
<tr>
<th>#</th>
<th>Matrix</th>
<th>m</th>
<th>n</th>
<th>nnz</th>
<th>flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>tp-6</td>
<td>142752</td>
<td>1014301</td>
<td>11537419</td>
<td>255 G</td>
</tr>
<tr>
<td>2</td>
<td>karted</td>
<td>46502</td>
<td>133115</td>
<td>1770349</td>
<td>258 G</td>
</tr>
<tr>
<td>3</td>
<td>EternityII_E</td>
<td>11077</td>
<td>262144</td>
<td>1572792</td>
<td>544 G</td>
</tr>
<tr>
<td>4</td>
<td>degme</td>
<td>185501</td>
<td>659415</td>
<td>8127528</td>
<td>591 G</td>
</tr>
<tr>
<td>5</td>
<td>cat_ears_4_4</td>
<td>19020</td>
<td>44448</td>
<td>132888</td>
<td>716 G</td>
</tr>
<tr>
<td>6</td>
<td>e18</td>
<td>24617</td>
<td>38602</td>
<td>156466</td>
<td>3399 G</td>
</tr>
<tr>
<td>7</td>
<td>flower_7_4</td>
<td>27693</td>
<td>67593</td>
<td>202218</td>
<td>4261 G</td>
</tr>
<tr>
<td>8</td>
<td>Rucci1</td>
<td>1977885</td>
<td>109900</td>
<td>7791168</td>
<td>12768 G</td>
</tr>
</tbody>
</table>
Relative maximum DAG size -- AMD 24 cores

![Graph showing the relative maximum DAG size for different numbers of threads. The x-axis represents the number of threads (1 to 8), and the y-axis shows the relative maximum DAG size. The graph indicates that the maximum DAG size increases as the number of threads increases, peaking at around 6 threads.](image-url)
Experimental results

Speedup -- AMD 24 cores

- qrm
- qrm_starpu
- avg.
- worst

number of threads

Journée des doctorants 2012. Toulouse, September 7th
Experimental results

Speedup -- AMD 24 cores

- spqr
- qrm_starpu

- best
- avg.
- worst

number of threads

Journée des doctorants 2012. Toulouse, September 7th
Experimental results

Relative performance compared to \textit{qrm} -- AMD 24 cores

- \texttt{qrm\_starpu}
- \texttt{spqr}

Matrix number

Journée des doctorants 2012. Toulouse, September 7th
Experimental results: conclusion

Regarding the scalability and factorization times:

- Better results than *spqr*:
  - Tree parallelism handled with Intel TBB and node parallelism delegated to multithreaded BLAS
  - Numerous synchronization points limiting the scalability

- Still poorer results than *qrm*:
  - Sophisticated scheduling policy with several optimizations
  - Tree pruning: the factorization of sub-trees with a relatively small computational weight is processed sequentially
  - Cost of the runtime system with small matrices
Methods and algorithms must be developed to exploit the potential of heterogeneous architectures:

- *communication avoiding* algorithms:
  
  heterogeneous architectures $\Rightarrow$ memory transfer must be avoided
Future work

Methods and algorithms must be developed to exploit the potential of heterogeneous architectures:

- *communication avoiding* algorithms:
  - heterogeneous architectures $\Rightarrow$ memory transfer must be avoided

- kernels adapted to heterogeneous architectures
  - complex emerging architectures $\Rightarrow$ need to re-think the existing algorithms
Methods and algorithms must be developed to exploit the potential of heterogeneous architectures:

- **communication avoiding** algorithms:
  
  heterogeneous architectures $\Rightarrow$ memory transfer must be avoided

- kernels adapted to heterogeneous architectures
  
  complex emerging architectures $\Rightarrow$ need to re-think the existing algorithms

- **combinatorial methods** for analysing and preparing the data (dynamically) provided to the runtime system
  
  large DAG $\Rightarrow$ cannot be statically handled by the runtime system
Future work

- Dynamic modifications of the DAG via composable/divisible tasks
Future work

- **Dynamic modifications of the DAG via composable/divisible tasks**

![Diagram](image)

- scheduling policies economical in terms of memory consumption
Thanks!
Questions?