Direct methods on GPU-based systems

Preliminary work towards a functioning code

A. Decollas and F. Lopez, Joint work with IRIT Toulouse, LaBRI / Inria Bordeaux, LIP / Inria Lyon

Sparse Days 2012. Toulouse, June 25th
Context of the work
F. Lopez @ IRIT-Toulouse
Evaluate the efficiency of modern runtime systems for heterogeneous and irregular workloads such as 
**Multifrontal solvers** on homogeneous, multicore architectures.

A. Decollas @ Inria-Bordeaux
Develop **dense linear algebra kernels** specific to sparse, direct solvers capable of achieving high efficiency on heterogeneous systems equipped with multiple CPUs and GPUs.

These two activities will ultimately be merged into a sparse, direct solver for accelerated multicore systems.
The multifrontal method

The multifrontal 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 $k$ rows/columns related to the pivots and all the other coefficients concerned by their elimination
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 coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix.

- **factorization**: the \( k \) pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - \( k \) rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent’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 coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix

- **factorization**: the $k$ pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - $k$ rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent’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 coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix.

- **factorization**: the $k$ pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - $k$ rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent’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 coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix.

- **factorization**: the $k$ pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - $k$ rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent’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 coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix.

- **factorization**: the $k$ pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - $k$ rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent’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 coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix.
- **factorization**: the *k* pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - *k* rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent’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 coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix.

- **factorization**: the $k$ pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - $k$ rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent’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 coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix.

- **factorization**: the $k$ pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - $k$ rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent’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 coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix

- **factorization**: the $k$ pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - $k$ rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent’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 coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix
- **factorization**: the $k$ pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - $k$ rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent’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 coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix.

- **factorization**: the $k$ pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - $k$ rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent’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)

$\Rightarrow$ specific algorithms
• 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 systems: 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
The multifrontal QR factorization of a sparse matrix $A = QR$ follows the pattern defined by the Cholesky factorization of the associated normal equations $A^T A = LL^T$ because of the equivalence of $R$ and $L$.

It shares most of the features of the multifrontal Cholesky algorithm apart from (most importantly):

- Frontal matrices are, in general, rectangular (both over or under-determined)
- Frontal matrices are fully factorized
- Contribution blocks are stacked and not summed
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:
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.
• 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:** some relatively small matrices from the UF collection

<table>
<thead>
<tr>
<th>Matrix</th>
<th>m</th>
<th>n</th>
<th>nnz</th>
<th>flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>degme</td>
<td>185501</td>
<td>659415</td>
<td>8127528</td>
<td>591 G</td>
</tr>
<tr>
<td>karted</td>
<td>46502</td>
<td>133115</td>
<td>1770349</td>
<td>258 G</td>
</tr>
<tr>
<td>flower_7_4</td>
<td>27693</td>
<td>67593</td>
<td>202218</td>
<td>4261 G</td>
</tr>
<tr>
<td>EternityII_E</td>
<td>11077</td>
<td>262144</td>
<td>1503732</td>
<td>544 G</td>
</tr>
<tr>
<td>cat_ears_4_4</td>
<td>19020</td>
<td>44448</td>
<td>132888</td>
<td>716 G</td>
</tr>
<tr>
<td>tp-6</td>
<td>142752</td>
<td>1014301</td>
<td>11537419</td>
<td>255 G</td>
</tr>
</tbody>
</table>
Experimental results

![Graph 1: AMD 24 cores -- degme](image1)

![Graph 2: AMD 24 cores -- karted](image2)
## Experimental results

### AMD 24 cores -- flower_7_4

<table>
<thead>
<tr>
<th># cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>12</th>
<th>18</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (sec.)</td>
<td>100</td>
<td>200</td>
<td>300</td>
<td>400</td>
<td>500</td>
<td>600</td>
<td>700</td>
</tr>
</tbody>
</table>

### AMD 24 cores -- EternityII_E

<table>
<thead>
<tr>
<th># cores</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (sec.)</td>
<td>12</td>
<td>4</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td>12</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

**Graphs:**

- **AMD 24 cores -- flower_7_4**
  - qrm
  - qrm_starpu
  - spqr

- **AMD 24 cores -- EternityII_E**
  - qrm
  - qrm_starpu
  - spqr

**Images:**

- AMD 24 cores -- flower_7_4
- AMD 24 cores -- EternityII_E
Experimental results

AMD 24 cores -- cat_ears_4_4

AMD 24 cores -- tp-6
Experimental results

Execution trace for the degme matrix:

Two main problems can be identified:

- Too much time is spent into tasks submission (in red). The issue is under investigation.
- At the moment, parent-child dependencies are not finely managed which means that it is not possible to start working on a node until all of its children are completely factorized.
Multifrontal Cholesky: front factorization on CPU-GPU hybrid systems
Bottom-up traversal of the elimination tree.
At each vertex (front):
- Assembling of contribution blocks from children
- Partial factorization of the frontal matrix
From elimination tree to frontal matrix

Bottom-up traversal of the elimination tree.
At each vertex (front):

- Assembling of contribution blocks from children
- Partial factorization of the frontal matrix
Tile Cholesky front factorization

• Derived from: *A class of parallel tiled linear algebra algorithms for multicore architectures*. Buttari et al., Parallel Comput., 2009

• Extension: Partial factorization of NPiv variables and computation of the Schur complement

• Use of a runtime system: StarPU
Tile Cholesky front factorization, \( NB|NPiv \) case

- NB: tile size
- NPiv: number of pivots
Tile Cholesky front factorization, $NB|NPiv$ case

- NB: tile size
- NPiv: number of pivots

![Diagram showing the tile Cholesky factorization with NB and NPiv dimensions.](image)

**KERNELS**

- Symetric tiles
- Up-to-date data
- potrf
- trsm
- trailing submatrix
Tile Cholesky front factorization, $NB|NPiv$ case

- **NB**: tile size
- **NPiv**: number of pivots

KERNELS

| Symetric tiles | Up-to-date data | potrf | syrk | trsm | gemm |
Tile Cholesky front factorization, $NB|NPiv$ case

- **NB**: tile size
- **NPiv**: number of pivots

![Diagram showing the relationship between NB and NPiv with a grid representation. The grid includes symbols for up-to-date data, current panel, and trailing submatrix.](diagram.png)
Task flow

- Fine granularity
- High concurrency
- Out-of-order execution
Multicore architecture

- 12 cores, 2.67 GHz Dual-socket Hexa-core Westmere Intel Xeon X5650 processor
- 12 MB L3 cache
- 36 GB RAM
Performance (multicore architecture)
Cholesky front factorization, general case (any NPiv)

NPiv

Up-to-date data
current pannel
trailing submatrix

KERNELS

Symetric tiles
Up-to-date data
current pannel
trailing submatrix
Cholesky front factorization, general case (any NPiv)
Cholesky front factorization, general case (any NPiv)
Performance (multicore architecture)
CPU-GPU hybrid architecture

- **12 cores, 2.67 GHz** Dual-socket Hexa-core Westmere Intel Xeon X5650 processor
- **12 MB** L3 cache
- **36 GB** RAM
- **3 NVIDIA Tesla M2070 (Fermi) GPUs**
Performance (CPU-GPU hybrid architecture) - $N=40K$

![Graph showing performance (GFlop/s) vs. NPIv for different numbers of GPUs (NGPUs).]
Execution trace (CPU-GPU hybrid architecture)
Conclusion, perspectives
• Preliminary work towards a functioning multifrontal code

• Not as efficient (for now) as codes designed for a specific architecture

• Performance portability across architectures

• Dense kernels to be released in the MAGMA library
Future work

- Robust solver (Cholesky assembly step, solution steps)
- Memory consumption (progressive task activation)
- Cluster of heterogeneous nodes
- Investigate explicit data dependencies management (DAGuE)
Future work

- Robust solver (Cholesky assembly step, solution steps)
- Memory consumption (progressive task activation)
- Cluster of heterogeneous nodes
- Investigate explicit data dependencies management (DAGuE)

Thanks!
Appendix
Validation (step 1)

- Factorization on nelim
Validation (step 1)

- Factorization on nelim

![Diagram](image_url)
Validation (step 1)

- Update the trailing submatrix

![Diagram of matrix with colored regions indicating different kernels: Symmetric tiles, Updated tiles, potrf, syrk, trsm, gemm.](image)
Validation (step 1)

- Update the trailing submatrix
• Finish factorization of "nelim" tile
Validation (step 2)

- Update the rest of the matrix
Validation (step 2)

- Factorize the rest of the matrix, as if NB | nelim