Architecture-aware Algorithms and Software for Peta and Exascale Computing

Jack Dongarra

University of Tennessee
Oak Ridge National Laboratory
University of Manchester
## November 2013: The TOP10

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National University of Defense Technology</td>
<td>Tianhe-2 NUDT, Xeon 12C 2.2GHz + IntelXeon Phi (57c) + Custom</td>
<td>China</td>
<td>3,120,000</td>
<td>33.9</td>
<td>17.8</td>
<td>1905</td>
</tr>
<tr>
<td>2</td>
<td>DOE / OS Oak Ridge Nat Lab</td>
<td>Titan, Cray XK7 (16C) + Nvidia Kepler GPU (14c) + Custom</td>
<td>USA</td>
<td>560,640</td>
<td>17.6</td>
<td>8.3</td>
<td>2120</td>
</tr>
<tr>
<td>3</td>
<td>DOE / NNSA L Livermore Nat Lab</td>
<td>Sequoia, BlueGene/Q (16c) + custom</td>
<td>USA</td>
<td>1,572,864</td>
<td>17.2</td>
<td>7.9</td>
<td>2063</td>
</tr>
<tr>
<td>4</td>
<td>RIKEN Advanced Inst for Comp Sci</td>
<td>K computer Fujitsu SPARC64 VIIIfx (8c) + Custom</td>
<td>Japan</td>
<td>705,024</td>
<td>10.5</td>
<td>12.7</td>
<td>827</td>
</tr>
<tr>
<td>5</td>
<td>DOE / OS Argonne Nat Lab</td>
<td>Mira, BlueGene/Q (16c) + Custom</td>
<td>USA</td>
<td>786,432</td>
<td>8.16</td>
<td>3.95</td>
<td>2066</td>
</tr>
<tr>
<td>6</td>
<td>Swiss CSCS</td>
<td>Piz Daint, Cray XC30, Xeon 8C + Nvidia Kepler (14c) + Custom</td>
<td>Switzerland</td>
<td>115,984</td>
<td>6.27</td>
<td>2.3</td>
<td>2726</td>
</tr>
<tr>
<td>7</td>
<td>Texas Advanced Computing Center</td>
<td>Stampede, Dell Intel (8c) + Intel Xeon Phi (61c) + IB</td>
<td>USA</td>
<td>204,900</td>
<td>2.66</td>
<td>3.3</td>
<td>806</td>
</tr>
<tr>
<td>8</td>
<td>Forschungszentrum Juelich (FZJ)</td>
<td>JuQUEEN, BlueGene/Q, Power BQC 16C 1.6GHz+Custom</td>
<td>Germany</td>
<td>458,752</td>
<td>5.01</td>
<td>2.30</td>
<td>2178</td>
</tr>
<tr>
<td>9</td>
<td>DOE / NNSA L Livermore Nat Lab</td>
<td>Vulcan, BlueGene/Q, Power BQC 16C 1.6GHz+Custom</td>
<td>USA</td>
<td>393,216</td>
<td>4.29</td>
<td>1.97</td>
<td>2177</td>
</tr>
<tr>
<td>10</td>
<td>Leibniz Rechenzentrum</td>
<td>SuperMUC, Intel (8c) + IB</td>
<td>Germany</td>
<td>147,456</td>
<td>2.90</td>
<td>3.42</td>
<td>848</td>
</tr>
</tbody>
</table>

500 Banking | HP | USA | 22,212 | .118 | 50 |
Accelerators (53 systems)
Top500 Performance Share of Accelerators

Fraction of Total TOP500 Performance

Year: 2006 to 2013
For the Top 500: Rank at which Half of Total Performance is Accumulated
Commodity plus Accelerator Today

Commodity
- Intel Xeon
  - 8 cores
  - 3 GHz
  - 8*4 ops/cycle
  - 96 Gflop/s (DP)

Accelerator (GPU)
- Nvidia K20X “Kepler”
  - 2688 “Cuda cores”
  - .732 GHz
  - 2688*2/3 ops/cycle
  - 1.31 Tflop/s (DP)

Interconnect
- PCI-X 16 lane
- 64 Gb/s (8 GB/s)
- 1 GW/s

192 Cuda cores/SMX
2688 “Cuda cores”
Linpack Efficiency
Linpack Efficiency
Linpack Efficiency
We are interested in developing Dense Linear Algebra Solvers

Retool LAPACK and ScaLAPACK for hybrid architectures
# A New Generation of DLA Software

<table>
<thead>
<tr>
<th>Software/Algorithms follow hardware evolution in time</th>
<th>LINPACK (70’s) (Vector operations)</th>
<th>Rely on</th>
<th>Level-1 BLAS operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPACK (80’s) (Blocking, cache friendly)</td>
<td></td>
<td>Rely on</td>
<td>Level-3 BLAS operations</td>
</tr>
<tr>
<td>ScaLAPACK (90’s) (Distributed Memory)</td>
<td></td>
<td>Rely on</td>
<td>PBLAS Mess Passing</td>
</tr>
</tbody>
</table>

## 2D Block Cyclic Layout

<table>
<thead>
<tr>
<th>Matrix point of view</th>
<th>Processor point of view</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 4 0 2 4 0 2 4</td>
<td>0 0 0 2 2 4 4 4 4</td>
</tr>
<tr>
<td>1 3 5 1 3 5 1 3 5</td>
<td>0 0 0 2 2 4 4 4 4</td>
</tr>
<tr>
<td>0 2 4 0 2 4 0 2 4</td>
<td>0 0 0 2 2 4 4 4 4</td>
</tr>
<tr>
<td>1 3 5 1 3 5 1 3 5</td>
<td>0 0 0 2 2 4 4 4 4</td>
</tr>
<tr>
<td>0 2 4 0 2 4 0 2 4</td>
<td>0 0 0 2 2 4 4 4 4</td>
</tr>
<tr>
<td>1 3 5 1 3 5 1 3 5</td>
<td>0 0 0 2 2 4 4 4 4</td>
</tr>
<tr>
<td>0 2 4 0 2 4 0 2 4</td>
<td>0 0 0 2 2 4 4 4 4</td>
</tr>
<tr>
<td>1 3 5 1 3 5 1 3 5</td>
<td>0 0 0 2 2 4 4 4 4</td>
</tr>
<tr>
<td>0 2 4 0 2 4 0 2 4</td>
<td>0 0 0 2 2 4 4 4 4</td>
</tr>
</tbody>
</table>

---

**MAGMA**

Hybrid Algorithms (heterogeneity friendly)

- hybrid scheduler
- hybrid kernels

Software/Algorithms follow hardware evolution in time

LINPACK (70’s) (Vector operations) Rely on Level-1 BLAS operations

LAPACK (80’s) (Blocking, cache friendly) Rely on Level-3 BLAS operations

ScaLAPACK (90’s) (Distributed Memory) Rely on PBLAS Mess Passing

New Algorithms (many-core friendly)

- a DAG/scheduler
- block data layout
- some extra kernels
MAGMA: LAPACK for GPUs

**MAGMA**
- Matrix algebra for GPU and multicore architecture
- To provide LAPACK/ScaLAPACK on hybrid architectures
- http://icl.cs.utk.edu/magma/

**MAGMA for CUDA, Intel Xeon Phi, and OpenCL**
- Hybrid dense linear algebra:
  - One-sided factorizations and linear system solvers
  - Two-sided factorizations and eigenproblem solvers
  - A subset of BLAS and auxiliary routines

**MAGMA developers & collaborators**
- UTK, UC Berkeley, UC Denver, INRIA (France), KAUST (Saudi Arabia)
- Community effort, similar to LAPACK/ScaLAPACK
**HYBRID ALGORITHMS**

MAGMA uses a hybridization methodology where algorithms of interest are split into tasks of varying granularity and their execution scheduled over the available hardware components. Scheduling can be static or dynamic. In either case, small non-parallelizable tasks, often on the critical path, are scheduled on the CPU, and larger more parallelizable ones, often Level 3 BLAS, are scheduled on the GPU.

**PERFORMANCE**

![Graph showing performance of MAGMA LU in DP on Kepler K20X]

- **Performance of MAGMA LU in DP on Kepler K20X**
  - **2 GPUs**: MAGMA
  - **1 GPU**: MAGMA
  - **CPU**: MKL

**FEATURES AND SUPPORT**

- **MAGMA 1.4** for **CUDA**
- **cIMAGMA 1.0** for **OpenCL**
- **MAGMA MIC 1.0** for **Intel Xeon Phi**

**Features**

- Linear system solvers
- Eigenvalue problem solvers
- MAGMA BLAS
- CPU Interface
- GPU Interface
- Multiple precision support
- Non-GPU-resident factorizations
- Multicore and multi-GPU support
- Tile factorizations with StarPU dynamic scheduling
- LAPACK testing
- Linux
- Windows
- Mac OS
### A New Generation of DLA Software

<table>
<thead>
<tr>
<th>Software/Algorithms follow hardware evolution in time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LINPACK (70’s)</strong> (Vector operations)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>LAPACK (80’s)</strong> (Blocking, cache friendly)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>ScaLAPACK (90’s)</strong> (Distributed Memory)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>PLASMA (00’s)</strong> New Algorithms (many-core friendly)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>MAGMA</strong> Hybrid Algorithms (heterogeneity friendly)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Parallelization of LU and QR.

Parallelize the update:
- Easy and done in any reasonable software.
- This is the 2/3n^3 term in the FLOPs count.
- Can be done efficiently with LAPACK+multithreaded BLAS.

\begin{align*}
    \text{dgetf2} & \quad \text{lu}(A^{(1)}) \\
    \text{dtrsm ( + dswp)} & \\
    \text{dgemm} & \\
    \text{dgetf2} & \quad \text{lu}(A^{(2)})
\end{align*}

Fork - Join parallelism
Bulk Sync Processing
Synchronization (in LAPACK LU)

- fork join
- bulk synchronous processing
Numerical program generates tasks and run time system executes tasks respecting data dependences.
High Performance Computing : Current Development

We are developing a strategy:

- That prioritizes the data-intensive operations to be executed by the accelerator

- That keep the memory-bound ones for the CPUs since the hierarchical caches with out-of-order superscalar scheduling are more appropriate to handle it.

- Moreover, in order to keep the accelerator busy, we redesign the kernels and propose dynamically guided data distribution to exploit enough parallelism to keep the accelerators and processors busy.
A runtime environment for the dynamic execution of precedence-constraint tasks (DAGs) in a multicore machine

- Translation
- If you have a serial program that consists of computational kernels (tasks) that are related by data dependencies, QUARK can help you execute that program (relatively efficiently and easily) in parallel on a multicore machine.
The Purpose of a QUARK Runtime

Objectives
- High utilization of each core
- Scaling to large number of cores
- Synchronization reducing algorithms

Methodology
- Dynamic DAG scheduling (QUARK)
- Explicit parallelism
- Implicit communication
- Fine granularity / block data layout

Arbitrary DAG with dynamic scheduling

Fork-join parallelism
Notice the synchronization penalty in the presence of heterogeneity.
QUARK

Shared Memory Superscalar Scheduling

FOR k = 0..TILES-1
  A[k][k] ← DPOTRF(A[k][k])
  FOR m = k+1..TILES-1
    A[m][k] ← DTRSM(A[k][k], A[m][k])
  FOR m = k+1..TILES-1
    A[m][m] ← DSYRK(A[m][k], A[m][m])
  FOR n = k+1..m-1
    A[m][n] ← DGEMM(A[m][k], A[n][k], A[m][n])

for (k = 0; k < A.mt; k++) {
  QUARK_CORE_dpotrf(...);
  for (m = k+1; m < A.mt; m++) {
    QUARK_CORE_dtrsm(...);
  }
  for (m = k+1; m < A.mt; m++) {
    QUARK_CORE_dsyrk(...);
    for (n = k+1; n < m; n++) {
      QUARK_CORE_dgemm(...)
    }
  }
}
1. Standard hybrid CPU-GPU implementation

**Algorithm 1**: Two-phase implementation of a one-sided factorization.

```
for $P_i \in \{P_1, P_2, \ldots, P_n\}$ do
  CPU:
  Receive Panel($P_i$)
  PanelFactorize($P_i$)
  Send Panel($P_i$)
  GPU:
  TrailingMatrixUpdate($A^{(i)}$)
```

factor panel $k$ then update $\Rightarrow$ factor panel $k+1$
Standard implementation without lookahead:

- Execution trace of the Cholesky factorization on a single socket CPU (Sandy Bridge) and a K20c GPU.

- We see that the computation on the CPU (e.g., the panel factorization) is not overlapped with the computation on the GPU.

- The algorithm looks like sequential, the only advantage is that the data extensive operations are accelerated by the GPU.
2. Introducing a lookahead panel to overlap CPU and GPU

Factor panel $k$ then update $\Rightarrow$ factor panel $k+1$ next panel

**Algorithm 2:** Two-phase implementation with a split update and explicit communication.

for $P_i \in \{P_1, P_2, \ldots, P_n\}$ do

**CPU:**
Receive Panel($P_i$)
PanelFactorize($P_i$)
Send Panel($P_i$)

**GPU:**
NextPanelUpdate(lookahead $P_{(i+1)}$) $\rightarrow$ goto CPU
TrailingMatrixUpdate($A^{(i)}$)
**New implementation with lookahead:**

- Execution trace of the Cholesky factorization on a single socket CPU (Sandy Bridge) and a K20c GPU.

- We see that the memory-bound kernel (e.g., the panel factorization) has been allocated to the CPU while the compute-bound kernel (e.g., the update performed by DSYRK) has been allocated to the accelerator.

- The advantage of such strategy is not only to hide the data transfer cost between the CPU and GPU but also to keep the GPU busy all the way until the end of execution.
3. Prioritizing critical path to provide more parallelism if needed

Algorithm 3: Two-phase implementation with a split update prioritizing critical path.

\[
\text{for } P_i \in \{ P_1, P_2, \ldots, P_n \} \text{ do}
\]

**CPU:**
- Receive Panel\((P_i)\)
- PanelFactorize\((P_i)\)
- Send Panel\((P_i)\)

**GPU:**
- for \( j \in \{ P_{i+1}, P_{i+2}, \ldots, P_n \} \) do
  - MatrixUpdate of block \( j(P(j)) \) with priority \( p - j \)

\]
High Performance Computing: current development

Prioritize the critical path:

- the panel factorization can be executed earlier. This will increase the lookahead depth that the algorithm exposes, increasing parallelism, so that there are more update tasks available to be executed by the device resources.

- This option has advantage when a lot of parallelism is needed especially for small sizes.
High Performance Computing: current development

1. Standard hybrid CPU-GPU implementation

Algorithm 1: Two-phase implementation of a one-sided factorization.

\[
\text{for } P_i \in \{P_1, P_2, \ldots, P_n\} \text{ do}
\]

- **CPU:**
  - Receive Panel\(P_i\)
  - PanelFactorize\(P_i\)
  - Send Panel\(P_i\)

- **GPU:**
  - TrailingMatrixUpdate\(A^{(i)}\)

\[\Rightarrow \text{factor panel } k \quad \text{then update} \quad \text{factor panel } k+1\]
High Performance Computing : current development

Resource Capability Weight :

- the advantage of such strategy is to keep all resources busy all the way until the end of execution.

- Careful management of the capability-weights ensures that the CPU does not take any work that would cause a delay to the GPU, since that would negatively affect the performance.
High Performance Computing : current development

- DPOTRF using CW (Kepler K20c)
- DPOTRF no CW (Kepler K20c)
- DPOTRF using CW (Fermi M2090)
- DPOTRF no CW (Fermi M2090)

Gflop/s vs Matrix size
High Performance Computing: current development

**Multiple GPU Case**

- Experiments with 6 GPUs

Scalability and performance of such implementation
Scalability and efficiency:

- snapshot of the execution trace of the Cholesky factorization on System A for a matrix of size 40K using six GPUs K20c.

- As expected the pattern of the trace looks compressed which means that our implementation is able to schedule and balance the tasks on the whole six GPUs devices.
magma_quark  DPOTRF  Kepler  K20c

Matrix size  Gflop/s

- DPOTRF 1 K20c

Graph showing the performance of DPOTRF on Kepler K20c with different matrix sizes.
magma_quark  DPOTRF Kepler  K20c

![Graph showing the performance of DPOTRF on Kepler K20c. The graph plots Gflop/s against matrix size (in thousands). Two lines are shown: one for DPOTRF 2 K20c and one for DPOTRF 1 K20c. The performance increases with matrix size for both lines, with DPOTRF 2 K20c generally outperforming DPOTRF 1 K20c.]
magma_quark  DPOTRF  Kepler  K20c

![Graph showing the performance of DPOTRF on various matrix sizes on Kepler K20c.](image-url)
magma_quark  DGEQRF  Kepler  K20c

![Graph showing matrix size vs Gflop/s for DGEQRF 1 K20c]
magma_quark  DGEQRF  Kepler  K20c

![Graph showing performance of DGEQRF 1 K20c, DGEQRF 2 K20c, and DGEQRF 3 K20c for different matrix sizes. The graph plots Gflop/s against matrix size, with matrix sizes ranging from 2k to 56k.]
magma_quark DGEQRF Kepler K20c

![Graph](chart.png)
magma_quark scalability DGEQRF Xeon-Phi

- QR 3 XeonPhi
- QR 2 XeonPhi
- QR 1 XeonPhi

Gflop/s vs Matrix size

- 2k, 4k, 6k, 8k, 12k, 16k, 20k, 24k, 28k, 32k, 36k, 40k
Major Changes to Software & Algorithms

- Must rethink the design of our algorithms and software
  - Another disruptive technology
    - Similar to what happened with cluster computing and message passing
  - Rethink and rewrite the applications, algorithms, and software
- Data movement is expense
- Flop/s are cheap, so are provisioned in excess
Summary

- **Major Challenges are ahead for extreme computing**
  - Parallelism $O(10^9)$
    - Programming issues
  - Hybrid
    - Peak and HPL may be very misleading
    - No where near close to peak for most apps
  - Fault Tolerance
    - Today Sequoia BG/Q node failure rate is 1.25 failures/day
  - Power
    - 50 Gflops/w (today at 2 Gflops/w)

- **We will need completely new approaches and technologies to reach the Exascale level**
Collaborators / Software / Support

- **PLASMA**
  http://icl.cs.utk.edu/plasma/

- **MAGMA**
  http://icl.cs.utk.edu/magma/

- **Quark (RT for Shared Memory)**
  http://icl.cs.utk.edu/quark/

- **PaRSEC** *(Parallel Runtime Scheduling and Execution Control)*
  http://icl.cs.utk.edu/parsec/

- Collaborating partners
  University of Tennessee, Knoxville
  University of California, Berkeley
  University of Colorado, Denver