Tuning Basic Linear Algebra Routines for Hybrid CPU+GPU Platforms

Gregorio Bernabé, Luis P. García, Javier Cuenca and Domingo Giménez

Universidad de Murcia/Universidad Politécnica de Cartagena
Scientific Computing and Parallel Programming Group

International Conference on Computational Science
June 10-12, 2014
Due to the omnipresence of multicore systems with GPU accelerators:

- Necessary **software optimization techniques** to benefit from the potential of the CPU+GPU system
- **Modelling** the execution time of the routine and apply some **empirical approach** to study the behaviour

**In this work:**

- Empirical auto-tuning technique for a **basic** hybrid linear algebra **kernel**: methodology for installation and modelling
- How to **use** the basic auto-tuned **kernel** in a **higher** level **routine**. LU factorization.

Achieves optimum load balance between GPUs and CPUs when they are performing linear algebra routines
1. Introduction

2. Motivation

3. Auto-tuning a multi-device matrix multiplication

4. Auto-tuning a multi-device LU factorization by blocks

5. Conclusions and future research
Outline

1. Introduction

2. Motivation

3. Auto-tuning a multi-device matrix multiplication

4. Auto-tuning a multi-device LU factorization by blocks

5. Conclusions and future research
Motivation

- Autotuning technique for achieving optimum load balance between GPUs and CPUs in basic linear algebra routines
- Matrix multiplication kernel, the basic idea is to carry out a matrix-multiplication simultaneously on both GPU and CPU cores.
- Overlap the multi-device (CPU+GPU) computations and data transfers

\[
\text{DGEMM in CPU+GPU}
\]

\[
\text{DGEMM in CPU and GPU}
\]

\[
C = \alpha AB + \beta C \Rightarrow C = \alpha(AB_1 + AB_2) + \beta(C_1 + C_2)
\]

- \(\alpha AB_1 + \beta C_1\) can be performed in GPU and \(\alpha AB_2 + \beta C_2\) in CPU

\[
\text{Distribution between GPU and CPU}
\]

\[
N = N_{\text{gpu}} + N_{\text{cpu}} \text{ depend of } N, \text{ relative speed of GPU and CPU, number of cores in the system}
\]
Motivation

DGEMM CPU+GPU

// Asynchronous transfer requires pinned host memory
cudaMallocHost((void **) &h\*A, sizeof(double)*szeA);
// Copy async host memory to device
cublasSetMatrixAsync(M, K, h\*A, d\*A, ...);
cublasSetMatrixAsync(K, N\*gpu, h\*B+ldb*N\*cpu, d\*B, ...);
// Have GPU do C\*1 = AxB\*1
cublasDgemm(M, N\*gpu, K, d\*A, d\*B, d\*C, ...);
// Copy async results from device to host
cublasGetMatrixAsync(M, N\*gpu, d\*C, ldc, h\*C+ldc*N\*cpu, ...);
// Have CPU do C\*2 = AxB\*2
dgemm\*(&M, &N\*cpu, &K, h\*A, h\*B+ldb*N\*gpu, h\*C+ldc*N\*gpu, ...);

- GPU part: CUBLAS, MAGMA, CULA Tools
- CPU part with multithread BLAS: MKL, GotoBLAS, ATLAS
- Computations GPU-CPU are overlapped and data transfers GPU-CPU are performed asynchronously to achieve the maximum performance
- To reduce the data transfer time CPU-GPU, we use the pinned memory mechanism provided by CUDA
Outline

1. Introduction
2. Motivation
3. Auto-tuning a multi-device matrix multiplication
4. Auto-tuning a multi-device LU factorization by blocks
5. Conclusions and future research
Empirical modelling of the execution time

**General scheme empirical modelling**: $N_{CPU}$ and $N_{GPU}$

**INSTALLATION**

- **Installation Set**: \{384, 1152, \ldots, 8064\}
- $N_{CPU} = N_{CPU} + \Delta N_{CPU}$
- $N_{GPU} = N - N_{CPU}$

- **Execution**

**LEAST_SQUARE**

- $T_{comu}(n) = t_s + nt_w$
- $T_{comu_{h2d}}$ and $T_{comu_{d2h}}$
- $T_{sh2d}, t_{wh2d}$ and $t_{sd2h}, t_{wd2h}$

- $T_{dgemm}(m, n) = k_1 m^2 n + k_2 m^2 + k_3 m$
- $T_{dgemm_{gpu}}(m, n)$ and $T_{dgemm_{cpu}}(m, n)$
- $k_i_{gpu}$ and $k_i_{cpu}$

- $T_{EXEC} = \max (T_{dgemm_{cpu}} + \gamma T_{comu}, T_{dgemm_{gpu}} + T_{comu})$

**The model of the execution time of the hybrid DGEMM routine**

- $\gamma$: overlap of CPU computation and data transfer CPU-GPU. Obtained experimentally for a particular system $\gamma \in [0, 1]$

- **Experiments** with $M \in \text{Installation Set}$. Initial value for $N_{CPU} = 0$

- The value of $N_{CPU}$ is increased by a predetermined amount until the modelled execution time exceeds by a threshold the previous lowest modeled execution time $\frac{T_{EXEC} - T_{MIN}}{T_{MIN}} > \gamma h$
Installation of the hybrid *dgemm* routine

**General scheme empirical modelling** ⇒ *N_CPU* and *N_GPU*

**INSTALLATION**

\[ \text{Installation Set} \{384, 1152, \ldots, 8064\} \]

\[ \text{N_CPU} = \text{N_CPU} + \Delta \text{N_CPU} \]

\[ \text{N_GPU} = \text{N} - \text{N_CPU} \]

\[ T_{\text{comu}}(n) = t_s + nt_w \]

\[ T_{\text{comu}_{h2d}} \text{ and } T_{\text{comu}_{d2h}} \]

\[ t_{s_{h2d}}, t_{w_{h2d}} \text{ and } t_{s_{d2h}}, t_{w_{d2h}} \]

\[ T_{\text{dgemm}}(m, n) = k_1m^2n + k_2m^2 + k_3m \]

\[ T_{\text{dgemm}_{gpu}}(m, n) \text{ and } T_{\text{dgemm}_{cpu}}(m, n) \]

\[ k_{i_{gpu}} \text{ and } k_{i_{cpu}} \]

**LETAST_SQUARE**

\[ T_{\text{EXEC}} = \max (T_{\text{dgemm}_{cpu}} + \gamma T_{\text{comu}}, T_{\text{dgemm}_{gpu}} + T_{\text{comu}}) \]

**Installation**

- Estimates the time to transfer *n* bytes CPU-GPU
- Obtains *t_s* (the latency of sending the first byte) and *t_w* (the time required to send each subsequent byte)
- Estimated **linear regression** over experimental results for CUDA routines *cublasSetMatrixAsync* and *cublasGetMatrixAsync*

Bernabé et al. (SCPPG)  gbernabe@um.es  ICCS / June 10-12, 2014  9 / 24
Installation of the hybrid dgemm routine

**General scheme empirical modelling** \(\Rightarrow\) **\(N\text{\_CPU}\) and \(N\text{\_GPU}\)**

**INSTALLATION**

\[
N\text{\_CPU} = N\text{\_CPU} + \Delta N\text{\_CPU} \\
N\text{\_GPU} = N - N\text{\_CPU}
\]

**Installation Set**

\[\{384, 1152, \ldots, 8064\}\]

**Execution**

\[\text{hybrid\_DGEMM}(., M, N, K, A, LDA, B, LDB, C, LDC, B, LDB, N\text{\_CPU})\]

**LEAST\_SQUARE**

\[
T_{\text{comu}}(n) = t_s + nt_w \\
T_{\text{comu\_h2d}}\text{ and } T_{\text{comu\_d2h}} \\
t_{s\_h2d}, t_{w\_h2d} \text{ and } t_{s\_d2h}, t_{w\_d2h}
\]

\[
T_{\text{dgemm}}(m, n) = k_1m^2n + k_2m^2 + k_3m \\
T_{\text{dgemm\_gpu}}(m, n) \text{ and } T_{\text{dgemm\_cpu}}(m, n) \\
k_i_{\text{\_gpu}} \text{ and } k_i_{\text{\_cpu}}
\]

\[
T_{\text{EXEC}} = \max (T_{\text{dgemm\_cpu}} + \gamma T_{\text{comu}}, T_{\text{dgemm\_gpu}} + T_{\text{comu}})
\]

**Installation**

- Estimation of \(k_i\): **least-square** using the experimental results of simple benchmarks for the basic routines **dgemm** and **cublasDgemm** over specified data in the **Installation\_Set**

The benchmarks obtain the running times of the basic operations with the data storage and access scheme used in the hybrid routine
Installation of the hybrid \texttt{dgemm} routine

\section*{Computational systems}

- \textbf{12CK20}: is a shared-memory system with two hexa-cores (12 cores) Intel Xeon E5-2620 and a GPU device Tesla K20c (based on Kepler Architecture) with 4800 Mbytes of Global Memory and 2496 CUDA cores (13 Streaming Multiprocessors and 192 Streaming Processors)

\section*{Installation}

- It has been empirically tested that with $\gamma = 1$ is best predicts the time cost for the computational system \textbf{12CK20}

\begin{equation}
T_{\text{exec}} = \max \left( T_{\text{dgemm cpu}}, T_{\text{dgemm gpu}} \right) + T_{\text{comu}}
\end{equation}

- The reason is that the CPU is not idle during the copy of matrices $A$ and $B$ from CPU to GPU

- The average deviation between the modelled time and the measured time for problem sizes in the \textit{Installation Set} ranges from:
  - 4.14\% for medium and large matrix size
  - 11.44\% for small matrix sizes
Experimental results for the hybrid \texttt{dgemm} routine

<table>
<thead>
<tr>
<th>$n$</th>
<th>Model $N_{CPU}$</th>
<th>time</th>
<th>OPTIMUM $N_{CPU}$</th>
<th>time</th>
<th>Deviation (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>768</td>
<td>0</td>
<td>0.0036</td>
<td>0</td>
<td>0.0036</td>
<td>0.00</td>
</tr>
<tr>
<td>1536</td>
<td>48</td>
<td>0.0199</td>
<td>0</td>
<td>0.0171</td>
<td>16.61</td>
</tr>
<tr>
<td>2304</td>
<td>224</td>
<td>0.0424</td>
<td>240</td>
<td>0.0411</td>
<td>3.14</td>
</tr>
<tr>
<td>3072</td>
<td>384</td>
<td>0.0846</td>
<td>336</td>
<td>0.0842</td>
<td>0.46</td>
</tr>
<tr>
<td>3840</td>
<td>512</td>
<td>0.1459</td>
<td>512</td>
<td>0.1459</td>
<td>0.00</td>
</tr>
<tr>
<td>4608</td>
<td>640</td>
<td>0.2359</td>
<td>640</td>
<td>0.2359</td>
<td>0.00</td>
</tr>
<tr>
<td>5376</td>
<td>768</td>
<td>0.3562</td>
<td>800</td>
<td>0.3558</td>
<td>0.10</td>
</tr>
<tr>
<td>6144</td>
<td>896</td>
<td>0.5110</td>
<td>960</td>
<td>0.5100</td>
<td>0.18</td>
</tr>
<tr>
<td>6912</td>
<td>1008</td>
<td>0.7093</td>
<td>1072</td>
<td>0.7019</td>
<td>1.06</td>
</tr>
<tr>
<td>7680</td>
<td>1136</td>
<td>0.9618</td>
<td>1200</td>
<td>0.9375</td>
<td>2.59</td>
</tr>
<tr>
<td>8448</td>
<td>1264</td>
<td>1.2305</td>
<td>1280</td>
<td>1.2255</td>
<td>0.41</td>
</tr>
<tr>
<td>9216</td>
<td>1376</td>
<td>1.9682</td>
<td>1280</td>
<td>1.5803</td>
<td>24.55</td>
</tr>
<tr>
<td>9984</td>
<td>1504</td>
<td>2.1745</td>
<td>1280</td>
<td>2.1573</td>
<td>0.80</td>
</tr>
<tr>
<td>10572</td>
<td>1616</td>
<td>2.3111</td>
<td>1552</td>
<td>2.3101</td>
<td>0.04</td>
</tr>
<tr>
<td>11520</td>
<td>1744</td>
<td>3.3041</td>
<td>1392</td>
<td>3.0419</td>
<td>8.62</td>
</tr>
</tbody>
</table>

- Table for different matrix size in a Validation Set $\Rightarrow$ Execution time \texttt{dgemm} with optimum selection of $N_{CPU}$ and the selection provided by the empirical model

- $N_{CPU}$ is well predicted only in 3 of 15 cases. But the $N_{CPU}$ selected is very close to the optimum

- Not a great influence on the mean of the relative deviation from the optimum. Value of approximately 4%
Experimental results for the hybrid \texttt{dgemm} routine

![Graph showing matrix multiplication performance](image)

**GFLOPS average values obtained in 12CK20**

- The improvement is similar to that obtained with the optimum distribution (\textit{Hybrid DGEMM Optimum}), and very close to the addition of GFLOPS that can be obtained ideally working with MKL \texttt{dgemm} and CUBLAS \texttt{dgemm} separately (\textit{MKL+CUBLAS})
Outline

1. Introduction

2. Motivation

3. Auto-tuning a multi-device matrix multiplication

4. Auto-tuning a multi-device LU factorization by blocks

5. Conclusions and future research
Auto-tuning a multi-device LU factorization by blocks

Auto-tuning higher level routines

- Application of the methodology (Hybrid dgemm routine) to a higher level routine that use an auto-tuned multi-device kernel
- An LU factorization is used to illustrate the methodology
  - The same technique can be applied with other higher level routines: QR, Cholesky, etc
- The implementation has the same scheme as the LAPACK right-looking block LU algorithm (routine dgetrf)
A schema for the multi-device LU factorization by blocks

Multi-device LU implementation

- CPU kernel `dgetf2` for the panel factorization replaced `dgetrf` CPU kernel
- The triangular solve CPU kernel `dtrsm` replaced `cublasDtrsm` GPU kernel
- CPU kernel `dgemm` replaced by a hybrid (GPU+CPU) `dgemm`
- The auto-tuning is used for searching the best distribution of the work in the hybrid `dgemm` routine at each step of the LU factorization
Empirically Modelling

**Empirically Modelling hybrid LU routine**

- The values of the coefficients $k_i$ for the multiplication on GPU and for the multiplication on CPU are obtained as described previously. But taking into account that $m = n \gg b$.

- The performance improvement is greater than considering $m = n = k$, as further discussed in the experimental results section.

**General scheme empirical modelling**

\[
\text{INSTALLATION}\]

- **Installation Set**
  \[
  \{384, 1152, \ldots, 8064\}
  \]
- **N_CPU** = **N_CPU** + $\Delta$**N_CPU**
- **N_GPU** = **N** - **N_CPU**

\[
T_{\text{comu}}(n) = t_s + nt_w\]
\[
T_{\text{comu},h2d} \text{ and } T_{\text{comu},d2h}
\]
\[
t_{h2d}, \ t_{w,h2d} \text{ and } t_{d2h}, \ t_{w,d2h}
\]

\[
T_{\text{dgemm}}(m,n) = k_1m^2n + k_2m^2 + k_3m
\]
\[
T_{\text{dgemm},gpu}(m,n) \text{ and } T_{\text{dgemm},cpu}(m,n)
\]
\[
k_{i,\text{gpu}} \text{ and } k_{i,\text{cpu}}
\]

\[
T_{\text{EXEC}} = \max (T_{\text{dgemm},cpu} + \gamma T_{\text{comu}}, T_{\text{dgemm},gpu} + T_{\text{comu}})
\]
Installation of the hybrid LU routine

Computational systems

- **12CK20**: is a shared-memory system with two hexa-cores (12 cores) Intel Xeon E5-2620 and a GPU device Tesla K20c (based on Kepler Architecture) with 4800 Mbytes of Global Memory and 2496 CUDA cores (13 Streaming Multiprocessors and 192 Streaming Processors).

- **12CC2075**: is a shared-memory system with two hexa-cores (12 cores) Intel Xeon E5-2620, 2.00GHz, 32 GB of shared-memory and a GPU device Fermi Tesla C2075 with 5375 MBytes of Global Memory and 448 CUDA cores (14 Streaming Multiprocessors and 32 Streaming Processors).
Installation of the hybrid LU routine

Optimal case

Illustration of an optimal case, in which CPU and GPU overlapping the communication, the computation and complete their work at the same time in each step of the LU factorization

Execution time model

Empirically tested that the CPU work is overlapped with work on the GPU and the data transfers. Equation best predict the time cost for the computational systems

\[ T_{\text{exec}} = \max(T_{\text{dgemm_cpu}} + T_{\text{comu_cpu}}, T_{\text{dgemm_gpu}} + T_{\text{comu_gpu}}) \] (1)
Experimental results for the hybrid LU routine

We compared three versions of the LU. The Validation Set \( \neq \) Installation Set

- cpuLU: Version that calls to the CPU kernels from the BLAS implementation in the Intel MKL
- gpuLU: calls to the cudatasDgemm GPU kernel
- cpugpuLU: calls to the auto-tuning hybrid dgemm routine

<table>
<thead>
<tr>
<th>Version</th>
<th>12CC2075</th>
<th>12CK20</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpuLU</td>
<td>63.54</td>
<td>85.01</td>
</tr>
<tr>
<td>gpuLU</td>
<td>110.16</td>
<td>146.59</td>
</tr>
<tr>
<td>cpugpuLU</td>
<td>133.02</td>
<td>159.02</td>
</tr>
</tbody>
</table>

The auto-tuning methodology to use the CPUs in conjunction with the GPU improves the performance
Experimental results for the hybrid LU routine

Deviation in % of the GFLOPS achieved for cpugpuLU with respect to gpuLU. Validation Set $\neq$ Installation Set

Different strategies are used for selecting the value of $N_{CPU}$

- Average GFLOPS achieve with CUBLAS and MKL (CPU-GPU GFLOPS)
- Model obtained for the multiplication of square matrices ($m = n = k$)
- Model for the matrix multiplication used in the LU factorization ($m = n \gg k$). Outperforms always the other methods.
Outline

1. Introduction
2. Motivation
3. Auto-tuning a multi-device matrix multiplication
4. Auto-tuning a multi-device LU factorization by blocks
5. Conclusions and future research
Conclusions and future research

- An auto-tuning method is considered to obtain a balanced distribution of the work to execute LAR in CPU+GPU systems.
- The method used a model (theoretical-experimental) to search the best distribution of the work.
- The methodology is applied to a basic kernel and the proposal is studied for a higher level routine.
- The methodology seems to be an appropriate approach to lead to an optimum utilization of CPU+GPU systems.
- Now: applying the same technique to other high level routines (QR, Cholesky).
- In the future...: Extend the work to more complex platforms (cluster of nodes with multicore CPUs, multi GPUs, Intel Phi).
Tuning Basic Linear Algebra Routines for Hybrid CPU+GPU Platforms

Gregorio Bernabé, Luis P. García, Javier Cuenca and Domingo Giménez

Universidad de Murcia/Universidad Politécnica de Cartagena
Scientific Computing and Parallel Programming Group

International Conference on Computational Science
June 10-12, 2014