Computational Science for Undergraduate Research Experience (CSURE) 2014

RUNTIME SYSTEMS AND OUT-OF-CORE CHOLESKY FACTORIZATION ON THE INTEL XEON PHI SYSTEM

Students: Allan Richmond Razon Morales (GWU) and Chong Tian (CUHK)
Mentors: Dr. Kwai Wong (UT), Dr. Eduardo D’Azevedo (ORNL)
Collaborators: Dr. Shiquan Su (NICS), Dr. Asim YarKhan (UT), Ben Chan
Visual Overview

- PLASMA - dense algebra algorithms
- QUARK - multithreading and task management
- Intel MKL Library - optimized math library for comparison
Basic Specifications:
- Beacon: 157.3 TFLOPS (peak)
- 48 compute nodes
- Each node has access to four Intel Xeon Phi co-processors 5110P (MIC) and two 8-core Intel Xeon E5-2670 processors

Goals:
- Compare different runtime systems that can be run on the Intel Xeon Phi System
- Utilize QUARK within this system
- Optimize the QUARK performance tests to see if the program can be scaled efficiently

Performance testing was conducted on the host processor and its coprocessors through its normal execution.
2 x Intel Xeon Processor E5-2670
- 16 cores per node (8 per processor)
- 2.600 GHz Clock Speed
- 256 GB RAM

- **Pro**: More Memory (8x more)
- **Con**: Less Computational Power

4 x Intel Xeon Phi Coprocessor 5110P
- 60 cores
- 1.053 GHz Clock Speed
- 8 GB RAM

- **Pro**: More Computational Power (more cores)
- **Con**: Less Memory
Intel Xeon Architecture
(Offload Comparison Example)

```
#pragma offload target (mic)
#pragma omp parallel for reduction(+:pi)
```
Host: Normal Execution through host processor (compute node)
Native: Execution runs only directly on the co-processor (MIC)
Offload: Run on the host processor and then “offloads” dense calculations to the co-processor (Ideal for the OOC algorithm)
Runtime Systems

- Understanding each programming environment
  - Nested-For Loop Matrix Multiplication (MM) – QUARK
  - DGEMM – PLASMA, Intel MKL
  - Cholesky – Intel MKL

- All modes of executions were considered and tested
OOC Cholesky Using Dynamic Scheduling

- Cholesky Factorization
- Task DAG and QUARK
- OOC algorithm
- Further goals
Step 1: \( L_{11} \leftarrow \text{cholesky}( A_{11} ) \),
Step 2: \( L_{21} \leftarrow A_{21} / L_{11}^T \),
Step 3: \( A_{22} \leftarrow A_{22} - L_{21} \ast L_{21}^T \),
Step 4: \( L_{22} \leftarrow \text{cholesky}( A_{22} ) \),

<Panel factorization>
<Trailing submatrix update>
General Cholesky Using Tile Operations

\[
\begin{array}{cc}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{array}
\]

\[
\begin{array}{cccc}
A_{00} & A_{01} & \ldots & A_{0k} \\
A_{10} & A_{11} & \ldots & \ldots \\
\ldots & \ldots & \ldots & \ldots \\
A_{k0} & & A_{kk} & \\
\ldots & & \ldots & \ldots \\
A_{n0} & & & A_{nn}
\end{array}
\]
for k=0…n-1
  for j=k…n-1
    for i=j…n-1 {
      if (i=j=k) potrf (A(i,j)^r, A(i,j)^w)
      if (i>j=k) trsm (A(i,j)^r, A(k,k)^r, A(i,j)^w)
      if (i=j>k) syrk (A(i,j)^r, A(i,k)^r, A(i,j)^w)
      if (i>j>k) gemm (A(i,j)^r, A(i,k)^r, A(j,k)^r, A(i,j)^w)
    }

Task Directed Acyclic Graph (DAG)
/1. dpotrf type: (k,k,k) /
if((j==k) && (i==j))
 {
    list[count].Node=assignlabel(i,j,k); /* Assign node labels */
    list[count].node=(i+1+j*n)+k*n*n;
    list[count].type='F';
    fprintf(fp, "$ld[label=~""\$(\$ld,\$ld,\$ld)\"]POTRF", color=brown];\n", list[count].node,i,j,k);

/* in-nodes */
if(k>0) list[count].in[0]=assignlabel(i,j,k-1);
/* Assign data dependencies, i.e. edges */
for(q=0;q<3;q++)
 {
    if (!((list[count].in[q].I==I-1)||(list[count].in[q].J==J-1)||(list[count].in[q].K==K-1))
    fprintf(fp, "$ld->\$ld;", (list[count].in[q].I+1+list[count].in[q].J*n+list[count].in[q].K*n*n), list[count].node);
 }

/* out-nodes */
if(k<n-1)
 {
    for(q=1;q<n-k;q++) list[count].out[q-1]=assignlabel(k+q,k,k); /* to (k+q,k,k) */
 }

/* Assign the rank */
fprintf(fp, "\(\text{rank}=\text{same}\); depth\$ld \$ld\);\n", (k+1), list[count].node); /* if (i,j,k) is a type F, */
/* then it's on depth 3k+1/
<table>
<thead>
<tr>
<th>8</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original A=</td>
<td></td>
</tr>
<tr>
<td>2.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>1.000000</td>
<td>2.000000</td>
</tr>
<tr>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td>1.000000</td>
<td>1.000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.414214</td>
</tr>
<tr>
<td>0.707107</td>
</tr>
<tr>
<td>0.707107</td>
</tr>
<tr>
<td>0.707107</td>
</tr>
<tr>
<td>0.707107</td>
</tr>
<tr>
<td>0.707107</td>
</tr>
<tr>
<td>0.707107</td>
</tr>
<tr>
<td>0.707107</td>
</tr>
<tr>
<td>Before Factorization</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td><strong>(0,0) block:</strong></td>
</tr>
<tr>
<td>2.000000 1.000000 1.000000</td>
</tr>
<tr>
<td>1.000000 2.000000 1.000000</td>
</tr>
<tr>
<td>1.000000 1.000000 2.000000</td>
</tr>
<tr>
<td><strong>(0,1) block:</strong></td>
</tr>
<tr>
<td>1.000000 1.000000 1.000000</td>
</tr>
<tr>
<td>1.000000 1.000000 1.000000</td>
</tr>
<tr>
<td>1.000000 1.000000 1.000000</td>
</tr>
<tr>
<td><strong>(0,2) block:</strong></td>
</tr>
<tr>
<td>1.000000 1.000000 0.000000</td>
</tr>
<tr>
<td>1.000000 1.000000 0.000000</td>
</tr>
<tr>
<td>1.000000 1.000000 0.000000</td>
</tr>
<tr>
<td><strong>(1,0) block:</strong></td>
</tr>
<tr>
<td>1.000000 1.000000 1.000000</td>
</tr>
<tr>
<td>1.000000 1.000000 1.000000</td>
</tr>
<tr>
<td>1.000000 1.000000 1.000000</td>
</tr>
</tbody>
</table>
void CORE_incore_dpotrf( Quark *quark )

void QUARK_incore_dpotrf( Quark *quark, Quark_Task_Flags *task_flags, int uplo, int n, double *A, int nb)

......

/*1. dpotrf type:(k,k,k)/*
if((j_==k_)&&(i_==j_))
{
    /*set task flags*/
    Quark_Task_Flags tflags=Quark_Task_Flags_Initializer;
    QUARK_Task_Flag_Set(&tflags,TASK_PRIORITY,3);
    /*Insert the dpotrf task*/
    QUARK_incore_dpotrf(quark,&tflags,(int)'L',NB,A2[i_][j_],NB);
    continue;
}

# Motivation: CPU vs Coprocessor

<table>
<thead>
<tr>
<th>CPU</th>
<th>Coprocessors (GPU, Intel MIC, etc.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slower than Coprocessors for some certain computations like dense matrix multiplication.</td>
<td>Much faster and more energy efficient</td>
</tr>
<tr>
<td>Larger memory size</td>
<td>Limited amount of device memory</td>
</tr>
<tr>
<td></td>
<td>Data movement is expensive</td>
</tr>
</tbody>
</table>
## Out-of-Core Structure

<table>
<thead>
<tr>
<th>Out-of-core part</th>
<th>In-core Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Loads Y panels to the device memory</td>
<td>Factorize the sub-matrix residing on device memory, in which “right-looking” update is involved.</td>
</tr>
<tr>
<td>2. Apply the update from the part already factorized., which is called “left-looking” update.</td>
<td></td>
</tr>
</tbody>
</table>
/*Out of core part:(starting from the A(k,k) tile)*/
/*O1.Send in Y-panel*/

for j=k:1:k+sizeY-1

/*Expected optimization 0:find the optimal Y size*/

for i=j:1:n
    H2D_Copy A(i,j) -> Y(i,j)

/*Expected optimization 1:Since the right part of the lower part of "A" shrinks. for the same amount of space dedicated to the Y panel, we may use a wider Y-panel to store as many tiles as possible*/
/*O2.Left looking update, if not the first Y-panel*/
/*Send factorized columns into X panel*/
for i=1:1:k-1
{
    for j=k:1:n
        H2D_Copy L(i,j)->X(j)
/*Expected optimization 2: Some factorized panels can be copied into X panel immediately before written back to CPU*/
    for q=k:1:k+sizeY-1
        for p=q:1:n
            if(p==q) dsyrk(Y(p,q),X(p))
            else dgemm(Y(p,q),X(p),X(q)\(^T\))
/*Expected optimization 3: Use double buffering — while one X panel is doing dgemm, the other can be reading data concurrently*/
OOC Cholesky pseudo code

/*In core part : similar to the general Cholesky factorization, except there are extra data movements, especially from Y panel to X panel or to CPU*/

/*Expected optimization 5: Perform all dpotrf() operations on CPU*/
Simple 4*4 OOC Cholesky DAG
Further Goals

1. Complete the code combining OOC algorithm and general Cholesky factorization.

2. Extend to multiple MPI processes case.

3. Extend to LU factorization with pivoting and QR factorization.
Expectations from Runtime Results

- Which mode of execution is the most scalable?
- Is there a threshold or condition where the performance begins to remain constant or even fails?
Breakdown for Testing Approach

Testing Routines:
1. QUARK MM
2. PLASMA DGEMM Tiled
3. Intel MKL DGEMM
4. Intel MKL SPOTRF (Cholesky Factorization)

Measuring GFLOPS/s: (“Giga” Floating Operations per second)
1. For matrix multiplication and DGEMM:
   \[
   \frac{2n^3}{10^9 \times \text{time}_{\text{avg}}}
   \]
2. For Cholesky Factorization (SPOTRF):
   \[
   \frac{\frac{1}{3}n^3}{10^9 \times \text{time}_{\text{avg}}}
   \]
### QUARK Matrix Multiplication: Multi-threaded Tiled Routine

#### Performance Test for QUARK Tiled Matrix Multiplication (HOST)

<table>
<thead>
<tr>
<th>NB</th>
<th>4 threads</th>
<th>8 threads</th>
<th>16 threads</th>
<th>32 threads</th>
<th>64 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>13.46281933</td>
<td>12.5576024</td>
<td>12.173018</td>
<td>9.01319071</td>
<td>8.1316164</td>
</tr>
</tbody>
</table>

#### Performance Test for QUARK Tiled Matrix Multiplication (MIC)

<table>
<thead>
<tr>
<th>NB</th>
<th>4 threads</th>
<th>8 threads</th>
<th>16 threads</th>
<th>32 threads</th>
<th>64 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.99663077</td>
<td>1.38587357</td>
<td>1.496806</td>
<td>1.63284111</td>
<td>1.63284111</td>
</tr>
<tr>
<td>250</td>
<td>1.2182064</td>
<td>1.6359261</td>
<td>1.749207</td>
<td>1.8856365</td>
<td>1.8856365</td>
</tr>
<tr>
<td>500</td>
<td>1.4397817</td>
<td>1.8574972</td>
<td>1.972778</td>
<td>2.1090627</td>
<td>2.1090627</td>
</tr>
<tr>
<td>1000</td>
<td>1.6613571</td>
<td>2.0790774</td>
<td>2.194358</td>
<td>2.3306579</td>
<td>2.3306579</td>
</tr>
</tbody>
</table>

---

#### Table with Results

<table>
<thead>
<tr>
<th>NB</th>
<th>4 threads</th>
<th>8 threads</th>
<th>16 threads</th>
<th>32 threads</th>
<th>64 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>13.46281933</td>
<td>12.5576024</td>
<td>12.173018</td>
<td>9.01319071</td>
<td>8.1316164</td>
</tr>
<tr>
<td>500</td>
<td>52.32566097</td>
<td>47.4777321</td>
<td>45.76333371</td>
<td>23.1449664</td>
<td>22.262155</td>
</tr>
<tr>
<td>1000</td>
<td>53.95652097</td>
<td>50.2472455</td>
<td>52.62076229</td>
<td>22.262155</td>
<td>21.3790614</td>
</tr>
</tbody>
</table>

---

#### Table with Results

<table>
<thead>
<tr>
<th>NB</th>
<th>4 threads</th>
<th>8 threads</th>
<th>16 threads</th>
<th>32 threads</th>
<th>64 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.99663077</td>
<td>1.38587357</td>
<td>1.496806</td>
<td>1.63284111</td>
<td>1.63284111</td>
</tr>
<tr>
<td>250</td>
<td>1.70272846</td>
<td>2.15796714</td>
<td>2.21691933</td>
<td>2.22034778</td>
<td>2.22034778</td>
</tr>
<tr>
<td>500</td>
<td>3.03245308</td>
<td>3.6981071</td>
<td>3.537532</td>
<td>3.36262222</td>
<td>3.36262222</td>
</tr>
<tr>
<td>1000</td>
<td>5.5189</td>
<td>6.481805</td>
<td>5.77946333</td>
<td>4.94149778</td>
<td>4.94149778</td>
</tr>
<tr>
<td>64 threads</td>
<td>9.96874769</td>
<td>11.3790614</td>
<td>9.487648</td>
<td>6.68409111</td>
<td>6.68409111</td>
</tr>
</tbody>
</table>
PLASMA DGEMM Tiled Routine: MIC vs HOST

Performance Test for PLASMA DGEMM TILE
NB = 128, 60 Threads

GFLOPS/second vs Matrix Size (NxN)
PLASMA DGEMM Tiled Routine: Different Tile Sizes

![Graph showing performance of PLASMA DGEMM Tiled Routine with different tile sizes. The graph plots GFLOPS/second against Matrix Size (NxN) for various tile sizes and platforms.]
Intel MKL Routine DGEMM: Modes of Execution

Performance Tests for Different Modes of Execution for the Intel MKL DGEMM routine

- **HOST**
- **OFFLOAD**
- **MIC**

Matrix Size (NxN) vs. GFLOPS/second
Intel MKL Routine DGEMM: Threading within MIC
MIC Environment Variables

- **OMP_NUM_THREADS:**
  - Each coprocessor has 60 cores
  - Beacon has 4 per node.
  - Therefore, maximum value is 240.

- **KMP_AFFINITY:**
  - Compact: Sequential Queuing
  - Balanced: Threads allocated evenly among cores
Intel MKL Routine DGEMM: MIC Environment Variables

Performance Test for MIC environment variables:
Intel MKL DGEMM routine (N = 7680)
Intel MKL Routine SPOTRF (Cholesky Factorization) Modes of Execution

Performance Tests for Cholesky Factorization: Intel MKL Routine (spotrf)
Intel MKL Routine SPOTRF (Cholesky Factorization) Modes of Execution

Performance Tests for Cholesky Factorization:
MIC Environment Variables and Threading
Intel MKL Routine (spotrf)
Serial and OpenMP—results barely suffice for comparison

- ~0.6 GFLOPS/second for Serial
- ~0.6 GFLOPS/second for OpenMP (MIC)
- ~1.2 GFLOPS/second for OpenMP (Host)
Optimize QUARK implementations (matrix multiplication, DGEMM) with additional OpenMP and Offloading directives to produce better performance.

Incorporate the OOC Cholesky Factorization into QUARK and implement onto Beacon.
Thinking about the Future: Documentation (In Progress)

Methodology:

[1] Basic Matrix Multiplication (3 nested for loops)
- QUARK implementation (almost done)
  - host + MIC
  - three different NB = 100, 250, 500
  - matrix size are increments of 500
- Standard C / MKL
  - in progress

[2] DGEMM
- PLASMA tiled
  - host + MIC
- Intel MKL
  - host, offload, and MIC
  - MIC environment variables are crucial for optimization

[3] CHOL (spotrf)
- Intel MKL
  - host, offload, and MIC
  - same deal with MIC environment variable

[4] In Progress --- Optimize Quark

General Execution:

There are a number of ways to run a program:

[1] bash script

The first method is self-explanatory and can be used to configure the environment variables. These directories use plenty of them, which end in ".sh".

With bash scripts, a basic knowledge of the Portable Batch System (PBS) documentation in order for further configurations such as the duration
Conclusions

- QUARK implementation needs to be optimized to better utilize the MIC’s computational power.

- Given the range of 500:15000 at steps of 500 for the PLASMA DGEMM trial, increasing the tile size yielded better performance but increasing the number of threads proved insignificant.

- As expected, Intel’s optimized MKL performs 2.96x better than PLASMA’s DGEMM on the MIC:
  
  - 828.96038 and 279.89 GFLOPS/s respectively

- After running a number of stress tests for the Intel MKL Cholesky factorization, the best result at 741.18587 GFLOPS/s was attained by using the maximum number of available cores (OMP_NUM_THREADS=240) and organizing these cores in a compact manner (KMP_AFFINITY=compact).
Betro, Vincent. *Beacon Quickstart Guide at AACE/NICS*


Kurzak, Jakub. *PLASMA/QUARK and DPLASMA/PaRSEC tutorial: ICL UT Innovative Computing Laboratory.*


Images were derived from Google Images or their respective source (i.e., Intel)
Acknowledgements

- Dr. Kwai Wong + Dr. Christian Halloy, the JICS CSURE REU coordinators
- University of Tennessee for access to Star1 and the NICS conference rooms
- Oak Ridge National Laboratory for remote access to Beacon
- Ben Chan, Dr. Kwai Wong (UT), Dr. Asim YarKhan (UT), Dr. Eduardo D’Azevedo (ORNL), Dr. Shiquan Su (NICS), and XSEDE Help for all the documentation and help to progress in this research
- The National Science Foundation (NSF) for funding the CSURE Program
For questions about QUARK or PLASMA, please contact Dr. YarKhan.

For questions about the CSURE program, please contact Dr. Wong.

For questions about how this research was conducted, please contact Allan Richmond (arrm93@gwu.edu) or Terrence (tc92321@hotmail.com).

For fast troubleshooting help for Beacon, the Intel Xeon Phi System, or general supercomputing tips, contact XSEDE help email “help@xsede.org”