Delivering Parallel Programmability to the Masses via the Intel MIC Ecosystem: A Case Study

Kaixi Hou, Hao Wang, and Wu-chun Feng

Department of Computer Science, Virginia Tech
Intel Xeon Phi in HPC

- In the Top500 list* of supercomputers ... 
  - 27% of accelerator-based systems use Intel Xeon Phi (17/62)
  
Top 10:

1. Tianhe-2
2. Titan
3. Sequoia
4. K computer
5. Mira
6. Piz Daint
7. Stampede
8. JUQUEEN
9. Vulcan
10. Cray XC30

Tianhe-2

* Released in June 2014

<table>
<thead>
<tr>
<th>Name</th>
<th>Rmax (petaflop/s)</th>
<th>Xeon Phi /Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tianhe-2</td>
<td>33.86</td>
<td>3</td>
</tr>
<tr>
<td>Stampede</td>
<td>5.17</td>
<td>2</td>
</tr>
</tbody>
</table>

Equipped with Xeon Phi
Intel Xeon vs. Intel Xeon Phi

- Less than 12 cores/socket
- Cores @ ~3GHz
- 256-bit vector units
- DDR3 80~100GB/s BW

- Up to 61 cores
- Cores @ ~1 GHz
- 512-bit vector units
- GDDR5 150GB/s BW
• x86 architecture and programming models

Yes. it’s easy to write and run programs on Phi.

... but optimizing performance on Phi is not easy!
Architecture-Specific Solutions

- Transposition in FFT
  - Reduce memory accesses via cross-lane intrinsics

- Swendsen-Wang multi-cluster algorithm
  - Maneuver the elements in registers via the data-reordering intrinsics

- Linpack benchmark
  - Reorganize the computation patterns and instructions via assembly code

If the optimizations are Xeon Phi-specific, the codes are not easy to write and portable.
Performance, Programmability, and Portability

• It’s more than performance ...  
  ... programmability and portability.

• Solution: directive-based optimizations + “simple” algorithmic changes.
  – @Cache
    • Blocking to create better memory access
  – @Vector Units
    • Pragmas + loop structure changes
  – @Many-cores
    • Pragmas
  – Find the optimal combination of parameters.
Outline

• Introduction
  – Intel Xeon Phi
  – Architecture-Specific Solutions

• Case Study: Floyd-Warshall Algorithm
  – Algorithmic Overview
  – Optimizations for Xeon Phi
    • Cache Blocking
    • Vectorization via Data-Level Parallelism
    • Many Cores via Thread-Level Parallelism
  – Performance Evaluation on Xeon Phi

• Conclusion
Case Study: Floyd-Warshall Algorithm

- All-pairs shortest paths (APSP) problem
- Algorithmic complexity: $O(n^3)$
- Algorithmic Overview
  Keep an increasing subset of intermediate vertices for each iteration → dynamic programming problem
A Quick Example

- $k$ means the newly added intermediate vertex in current iteration.

\[
\begin{array}{c|cccc}
  & a & b & c & d \\
\hline
a & 0 & 4 & - & 4 \\
b & - & 0 & - & 3 \\
c & 2 & 6 & 0 & 6 \\
d & 4 & 8 & 1 & 0 \\
\end{array}
\]

$k = 1$

$k = 2$

$k = 3$

$k = 4$
Issue in Caching: Lack of Data Locality
Issue in Caching: Lack of Data Locality
Issue in Caching: Lack of Data Locality

Default algorithm: data locality problem
Cache Blocking: Improve Data Reuse

Step 1
(k, k)

Step 2
(k, j)

Step 3
(i, j)

Intermediate vertices
Cache Blocking: Improve Data Reuse

Step 1
(k, k)

Step 2
(k, j)

Step 3
(i, j)

Intermediate vertices

\[ k_1 \quad k_2 \quad k_3 \quad k_4 \]

\[ i_1 \quad i_2 \quad i_3 \quad i_4 \]
Cache Blocking: Improve Data Reuse

Intermediate vertices

Step 1: \((k, k)\)

Step 2: \((k, j)\)

Step 3: \((i, j)\)

\[ k_1, k_2, k_3, k_4 \]

\[ i_1, i_2, i_3, i_4 \]

\[ j_1, j_2, j_3, j_4 \]
Cache Blocking: Improve Data Reuse

Step 1: \((k, k)\)

Step 2: \((k, j)\)

Step 3: \((i, j)\)

Intermediate vertices

k_1 \rightarrow k_2 \rightarrow k_3 \rightarrow k_4

i_1 \rightarrow i_2 \rightarrow i_3 \rightarrow i_4

j_1 \rightarrow j_2 \rightarrow j_3 \rightarrow j_4
Cache Blocking: Improve Data Reuse

Step 1: (k, k)
Step 2: (k, j)
Step 3: (i, j)

Intermediate vertices

k_1 k_2 k_3 k_4

i_1 i_2 i_3 i_4

j_1 j_2 j_3 j_4

k_x \rightarrow k_z \rightarrow j_y

i_x \rightarrow k_z \rightarrow k_y
Cache Blocking: Improve Data Reuse

Step 1
(k, k)

Step 2
(k, j)

Step 3
(i, j)

Intermediate vertices

\[ \begin{align*}
  k_1 & \quad k_2 & \quad k_3 & \quad k_4 \\
  j_1 & \quad j_2 & \quad j_3 & \quad j_4 \\
  i_1 & \quad i_2 & \quad i_3 & \quad i_4 \\
  j_1 & \quad j_2 & \quad j_3 & \quad j_4
\end{align*} \]
Cache Blocking: Improve Data Reuse

Step 1 (k, k)
Step 2 (k, j)
Step 3 (i, j)

Intermediate vertices

k_1, k_2, k_3, k_4
i_1, i_2, i_3, i_4
j_1, j_2, j_3, j_4

padding
Cache Blocking: Improve Data Reuse

Step 1
(k, k)

Step 2
(k, j)

Step 3
(i, j)

Intermediate vertices

k₁ k₂ k₃ k₄

j₁ j₂ j₃ j₄

i₁ i₂ i₃ i₄

j₁ j₂ j₃ j₄

k₁ k₂ k₃ k₄

padding

i₁ i₂ i₃ i₄

Step 2
(i, k)

Step 3
(i, j)
Cache Blocking: Improve Data Reuse

Step 1
(k, k)

Step 2
(i, k)

Step 2
(k, j)

Step 3
(i, j)

Intermediate vertices

Padding
Vectorization: Data-Level Parallelism

Distance Matrix

void update(int k0, int u0, int v0) {
    ...       pmat[u][v]=k;
}}}}

Version 1
Vectorization: Data-Level Parallelism

- Pragmas to guide the compiler to vectorize the loop:
  - #pragma vector always: vectorize the loop regardless of the efficiency
  - #pragma ivdep: ignore vector dependencies

Distance Matrix

- Version 1
- Bottom-right block
- Core computation

Primed Area

void update(int k0, int u0, int v0) {
    for(k=k0;k<MIN(k0+BLOCK_SIZE, size);k++) {
        for(u=u0;u<MIN(u0+BLOCK_SIZE,size);u++) {
            #pragma ivdep
            for(v=v0;v<MIN(v0+BLOCK_SIZE,size);v++) {
                if(dmat[u][v]>dmat[u][k]+dmat[k][v])
                    dmat[u][v]=dmat[u][k]+dmat[k][v];
                pmat[u][v]=k;
            }
        }
    }
}
Vectorization: Data-Level Parallelism

void update(int k0, int u0, int v0) {
    for(k=k0;k<MIN(k0+BLOCK_SIZE, size);k++) {
        for(u=u0;u<MIN(u0+BLOCK_SIZE,size);u++) {
            #pragma ivdep
            for(v=v0;v<MIN(v0+BLOCK_SIZE,size);v++) {
                if(dmat[u][v]>dmat[u][k]+dmat[k][v])
                    pmat[u][v]=k;
                dmat[u][v]=dmat[u][k]+dmat[k][v];
            }
            pmat[u][v]=k;
        }
    }
}

Bottom-right block

Core computation

• Pragmas to guide the compiler to vectorize the loop:
  – #pragma vector always: vectorize the loop regardless of the efficiency
  – #pragma ivdep: ignore vector dependencies
Vectorization: Data-Level Parallelism (Cont’d)

void update(int k0, int u0, int v0) {
    for(k=k0;k<MIN(k0+BLOCK_SIZE, size);k++){
        for(u=u0;u<(u0+BLOCK_SIZE);u++) {
            #pragma ivdep
            for(v=v0;v<(v0+BLOCK_SIZE);v++) {
                if(dmat[u][v]>dmat[u][k]+dmat[k][v])
                    dmat[u][v]=dmat[u][k]+dmat[k][v];
                pmat[u][v]=k;
            }
        }
    }
}

Distance Matrix

Bottom-right block
• Modify the boundary check conditions (u-loop & v-loop)
  – Extra computations but regular loop forms
• Keep boundary check condition (k-loop)
  – Where to fetch data

Version 3
Distance Matrix
Bottom-right block SIMD-friendly codes

Extra computations
Extra computations

Extra computations
Extra computations
Many Cores: Thread-Level Parallelism (TLP)

- OpenMP pragmas
  - A portable way to parallelize serial programs
  - Run-time specifications: thread number, thread affinity, etc.

- Utilize thread-level parallelism (TLP) in Xeon Phi
  - Apply OpenMP pragmas on loops of step 2 and step 3: most parallelism opportunities.
Optimization Challenges in Thread-Level Parallelism

• Many configurable parameters
  – Ex: block size, thread number, runtime scheduling policy, etc.

• Difficulty in finding an appropriate combination of parameters
  – Inter-dependency between parameters
  – Huge search space
Optimization Approach to Thread-Level Parallelism

- Starchart: Tree-based partitioning

A pool of samples
Format: \((p_1, p_2, ..., p_n) \rightarrow \text{performance}\)

\[ V = \text{Performance Variance} \]

\[ V_i = \text{Performance Variance} + \text{Performance Variance} \]

Select parameter value which creates \(\max(V - V_i)\)
Applying Starchart

- **Data Size**: ∈ {2000, 4000}
- **Thread Num**: ∈ {61, 122, 183, 244}
- **Block Size**: ∈ {16, 32, 48, 64}
- **Task Alloc**: ∈ {block, cyc1, cyc2, cyc3}

- **200 samples mean = 1.47 s**
  - **Data Size ∈ {2000}**
  - **97 samples mean = 0.72 s**
    - **Block Size ∈ {16, 32, 48}**
      - **70 samples mean = 0.60 s**
    - **Thread Num ∈ {61}**
      - **20 samples mean = 0.72 s**
    - **Task Alloc ∈ {block}**
      - **7 samples mean = 0.50 s**
  - **103 samples mean = 2.17 s**
    - **Data Size ∈ {4000}**
    - **21 samples mean = 3.20 s**
      - **Thread Num ∈ {61}**
        - **82 samples mean = 1.91 s**
    - **Block Size ∈ {32, 48, 64}**
      - **55 samples mean = 1.79 s**
    - **Thread Num ∈ {183, 244}**

- **5 parameters: 480 possible combinations**
Applying Starchart

Small data size

- 70 samples mean = 0.60 s
- Block Size ∈ {16, 32, 48}
- Thread Num ∈ {61}
- 20 samples mean = 0.72 s
- Task Alloc ∈ {block}

Large data size

- 21 samples mean = 3.20 s
- Block Size ∈ {64}
- Thread Num ∈ {61}

- 82 samples mean = 1.91 s
- Block Size ∈ {122, 183, 244}
- Thread Num ∈ {122, 183, 244}

- 27 samples mean = 1.05 s
- Block Size ∈ {16}
- Thread Num ∈ {61, 122}

- 55 samples mean = 1.79 s
- Block Size ∈ {32, 48, 64}
- Thread Num ∈ {183, 244}

- 7 samples mean = 0.56 s
- Task Alloc ∈ {cyc1, cyc2, cyc3}

- 43 samples mean = 0.57 s
- Task Alloc ∈ {cyc1, cyc2, cyc3}

- 7 samples mean = 1.96 s
- Task Alloc ∈ {block}

- 34 samples mean = 1.68 s
- Task Alloc ∈ {cyc1, cyc2, cyc3}
Applying Starchart

Choosing appropriate block size and thread num is most important!

- **Data Size**: ∈ {2000, 4000}
  - 200 samples mean = 1.47 s
  - 97 samples mean = 0.72 s
  - 103 samples mean = 2.17 s

- **Thread Num**: ∈ {61, 122, 183, 244}
  - 82 samples mean = 1.91 s
  - 21 samples mean = 3.20 s
  - 7 samples mean = 1.96 s
  - 34 samples mean = 1.68 s

- **Block Size**: ∈ {16, 32, 48, 64, 32, 48, 64}
  - 70 samples mean = 0.60 s
  - 27 samples mean = 1.05 s
  - 50 samples mean = 0.56 s
  - 14 samples mean = 1.99 s

- **Task Alloc**: ∈ {block, cyc1, cyc2, cyc3}
  - 7 samples mean = 0.50 s
  - 43 samples mean = 0.57 s
  - 7 samples mean = 1.96 s
  - 34 samples mean = 1.68 s

Applying Starchart is crucial for optimizing performance.
Applying Starchart

Parameters (small): 32/blockSize, 244/threadNum, block/taskAlloc, balanced/threadAffinity

Parameters (large): 32/blockSize, 244/threadNum, cyc/taskAlloc, balanced/threadAffinity
Outline

• Introduction
  – Intel Xeon Phi
  – Architecture-Specific Solutions

• **Case Study : Floyd-Warshall Algorithm**
  – Algorithmic Overview
  – Optimizations for Xeon Phi
    • Cache Blocking
    • Vectorization via Data-Level Parallelism
    • Many Cores via Thread-Level Parallelism
  – Performance Evaluation for Xeon Phi

• Conclusion
Performance Evaluation: Step-by-Step

- **Cache blocking**: 14% performance loss
  - Redundant computations induced in step 2 and step 3.
  - Boundary check conditions in the loop structures

- **Cache blocking with changes to loop structure**: 1.76x

- **Vectorization via SIMD pragmas**: 4.09x

- **Thread-level parallelism via OpenMP pragmas**: 38.98x

- **Overall**: 281.67x
Performance Evaluation: Scalability

- Baseline: OpenMP version of default algorithm
- Optimized (MIC) vs. Baseline: up to 6.39x
- Optimized (MIC) vs. Optimized (CPU): up to 3.2x
  - Peak performance ratio of MIC and CPU: 3.23x (2148 Gflops and 665.6 Gflops)
Performance Evaluation: Strong Scaling

- Balanced thread affinity:
  - 2x from 1 thread/core to 4 threads/core
- Other affinities:
  - Scatter: 2.6x
  - Compact: 3.8x
Conclusion

• CPU programs can be recompiled and directly run on Intel Xeon Phi, but achieving optimized performance requires a considerable effort.
  – Considerations: Performance, programmability, and portability

• We use directive-based optimizations and certain algorithmic changes to achieve significant performance gains for the Floyd-Warshall algorithm as a case study.
  – 6.4x speedup over a default OpenMP version of Floyd-Warshall on Xeon Phi.
  – 3.2x speedup over a 12-core multicore CPU (Sandy Bridge).

Thanks! Questions?